uc3m | Universidad Carlos III de Madrid

Grado Universitario en Ingeniería Informática
(2017-2018)

Trabajo Fin de Grado

Sistema de monitorización distribuido para aplicaciones basadas en MPI

Edgar Mijero Bonde

Tutor

Francisco Javier García Blas

11 de julio de 2018, Leganés

Título: Sistema de monitorización distribuido de aplicaciones basadas en MPI.

Autor: Edgar Mijero Bonde

Tutor: Francisco Javier García Blas

EL TRIBUNAL

Presidente: _____

Vocal: _____

Secretario: _____

Realizado el acto de defensa y lectura del Trabajo de fin de grado el día __ de _____ de 20__ en Leganés, en la Escuela Politécnica Superior de la Universidad Carlos III de Madrid, acuerda otorgarle la CALIFICACIÓN de

VOCAL

SECRETARIO PRESIDENTE

RESUMEN

Dado el creciente interés en la computación de altas prestaciones, y concretamente en la computación paralela, cada vez se están desarrollando más herramientas que facilitan la programación paralela, paradigma que hasta hace poco estaba reservado al ámbito de la investigación, la academia y algunas ciencias que requieren gran capacidad de cómputo.

Pese al desarrollo de nuevas herramientas de programación, es necesario ofrecer herramientas que permitan a los usuarios poder medir o analizar el rendimiento obtenido por sus aplicaciones. Si bien existen herramientas que permiten capturar métricas de la ejecución de aplicaciones paralelas, dichas herramientas generan trazas o registros que no son fácilmente manejables o interpretables por los usuarios en ausencia de una interfaz gráfica, a la hora de llevar a cabo determinadas operaciones de análisis.

Lo que ofrece el sistema distribuido desarrollado en este Trabajo Fin de Grado es la monitorización aplicaciones paralelas basadas en paso de mensajes. Además, permite almacenar los datos obtenidos en un formato cómodo, portable entre plataformas, accesible y altamente extensible.

En este Trabajo de Fin de Grado se explica el procedimiento seguido para la implementación de dicho sistema de monitorización de aplicaciones basadas en MPI. Con los datos recogidos es posible llevar a cabo un análisis del rendimiento alcanzado durante la ejecución de dichas aplicaciones.

Palabras clave: MPI, rendimiento, monitorización, análisis, sistema, distribuido

ABSTRACT

Given the increasing interest on high performance computing, concretely on parallel computing, the number of tools that make programming parallel applications easier keeps rising. Parallel computing is a programming paradigm that was, until recently, reserved to research, academia and some sciences that require great computing power.

Despite the development of new parallel programming tools, it is necessary to offer users tools that allow them to measure and analyze the performance of their applications.

Even though there are some tools that allow to capture metrics related to execution of parallel applications, those tools generate traces and logs that are not easily interpretable by the users in the absence of a graphic user interface when performing certain analysis operations.

What the developed distributed system offers is the monitoring of parallel applications based on message passing. Furthermore, the system allows the persistence of the collected execution data in a portable across platforms, accessible and extensible format.

In this Final Project the approach followed to implement the monitoring system for applications based on MPI. With the collected data is possible to make and analysis of the performance level achieved by those applications.

Keywords: MPI, system, distributed, analysis, performance, monitoring, parallel

AGRADECIMIENTOS

En primer lugar, quisiera agradecer a toda mi familia por el apoyo que me han dado siempre, sobre todo a mis padres, Eugenio y Trinidad por el increíble esfuerzo que han hecho para que todo esto sea posible.

Quisiera dar las gracias también a mis amigos, por la paciencia que han tenido y los ánimos que me han dado siempre.

Por último, a la universidad, a los compañeros y profesores, especialmente a Javier, por la paciencia que ha tenido tanto en clase como durante la realización de este proyecto.

ÍNDICE

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

1.- INTRODUCCIÓN

En este primer apartado se explicará brevemente el desarrollo de este proyecto, la motivación detrás del mismo, así como los objetivos que se esperan alcanzar con su realización.

1.1- MOTIVACIÓN

Con el aumento de la investigación y el desarrollo de paradigmas como el de computación paralela y la popularización de las GPUs, la computación de altas prestaciones (en inglés *High Performance Computing* o *HPC*) es un campo que ha ganado importancia en los últimos años.

Dada la creciente actividad en esta área y la importancia del rendimiento en este modelo de computación, es importante disponer de herramientas que permitan medir, analizar y optimizar el rendimiento en aplicaciones paralelas.

Como consecuencia de estas necesidades en cuanto a la computación paralela, ha aumentado el número de herramientas que facilitan el desarrollo de aplicaciones siguiendo este paradigma.

En computación paralela existen distintos modelos de programación y distintas arquitecturas hardware. Esta diversidad, hace complicado que se puedan desarrollar herramientas genéricas lo suficientemente potentes como para ser efectivas. Por ello, se ha decidido contribuir al conjunto de herramientas disponibles, creando una herramienta para monitorizar la ejecución de aplicaciones paralelas que implementadas en el modelo de programación paralela basado en el paso de mensajes.

La motivación tras la realización de este proyecto es, pues, la de desarrollar un sistema distribuido que ofrezca la capacidad monitorizar la ejecución de aplicaciones que basan su ejecución en el modelo de paso de mensajes, concretamente en aplicaciones que utilizan alguna implementación del estándar MPI. Esta monitorización se hace de cara a poder analizar posteriormente el rendimiento de dichas aplicaciones.

1.2.- OBJETIVOS

El objetivo de este Trabajo de Fin de Grado es el desarrollo de un sistema distribuido que permita monitorizar el rendimiento en aplicaciones basadas en MPI, así como mostrar los resultados obtenidos durante el análisis.

Para alcanzar estos objetivos, se procederá de la siguiente manera:

- Establecer un contexto, explicando los conceptos y tecnologías más relevantes utilizadas para la consecución de los objetivos.
- Presentar y comentar algunas herramientas disponibles que realizan una función similar.
- Explicar el funcionamiento de la aplicación utilizada para capturar datos acerca de los nodos en los que se ejecutan las aplicaciones y que son relevantes para el análisis.
- Extender dicha aplicación para incorporarlo al sistema distribuido.
- Implementar una biblioteca que permita recoger datos relativos a MPI durante los intercambios de mensajes entre los nodos de las aplicaciones.

1.3.- MARCO REGULADOR

En este apartado se mencionan las implicaciones legales que tiene la implementación de la herramienta de monitorización de aplicaciones.

La Ley Orgánica de Protección de Datos, abreviada *LOPD*, establece el procedimiento a seguir en cuanto al tratamiento de datos personales, de manera que se garantice el respeto de los derechos, el honor y la intimidad de las personas físicas.

En el Artículo 3.a del Título 1 de la LOPD se definen los datos personales como "*cualquier información concerniente a personas físicas identificadas o identificables*".

La biblioteca desarrollada captura y almacena información relativa a funciones de MPI invocadas durante la ejecución de la aplicación analizada. Ninguno de los datos recogidos (tamaño de los mensajes, tipo de datos, etcétera) hace referencia a información de carácter personal de acuerdo con la definición previa. Por lo tanto, la LOPD no es aplicable en este caso.

1.4.- ESTRUCTURA DEL DOCUMENTO

Este documento se divide en los siguientes capítulos:

- En el primero se ofrecen las definiciones de las ideas más importantes relacionadas con MPI y la computación paralela. Además, se mencionan herramientas con funcionalidades similares a la herramienta desarrollada para este proyecto.

- En el siguiente apartado, se describe de manera detallada el funcionamiento de la herramienta, se enuncian los requisitos y especificaciones de la misma y se definen las métricas a analizar.

- En tercer lugar, se especifica la estructura de la herramienta, decisiones de diseño tomadas de cara a la implementación de la herramienta y la manera de proceder para su utilización (compilación, ejecución, etcétera). Además, se especifican la configuración y herramientas utilizadas para el análisis de datos.

- En cuarto lugar, se definen las diferentes pruebas a realizar. También se ofrece información acerca del entorno de pruebas, así como de su configuración.

- A continuación, se detalla la planificación seguida y el presupuesto estimado para el desarrollo de la herramienta desarrollada para poder realizar el análisis de aplicaciones.

- Para finalizar, se muestran y comentan los resultados obtenidos y las conclusiones extraídas a partir de los mismos. También se proponen posibles funcionalidades para extender la herramienta o posibles líneas de investigación de cara futuro.

2.- ESTADO DEL ARTE

En este capítulo se explican las tecnologías y conceptos más relevantes para la realización de este Proyecto de Fin de Grado. En primer lugar, se explican algunos términos relacionados la computación paralela. Posteriormente se define qué es MPI y se muestran las implementaciones más utilizadas de este estándar. Finalmente se revisan las principales herramientas que realizan funciones similares (*profiling*) o complementarias a la herramienta desarrollada durante el proyecto, incluyendo una comparativa en cuanto a las funcionalidades que ofrece cada una de ellas.

2.1.- ARQUITECTURAS EN COMPUTACIÓN PARALELA

En este apartado se procede a explicar las arquitecturas utilizadas en computación paralela, concretamente: la arquitectura de memoria distribuida, de memoria compartida y la híbrida.

2.1.1.- MEMORIA DISTRIBUIDA

El en modelo de memoria distribuida, cada procesador dispone de su propia memoria privada. Puesto que las regiones de memoria de cada proceso son independientes, es tarea del programador gestionar tanto la sincronización como el intercambio de datos locales entre procesos.

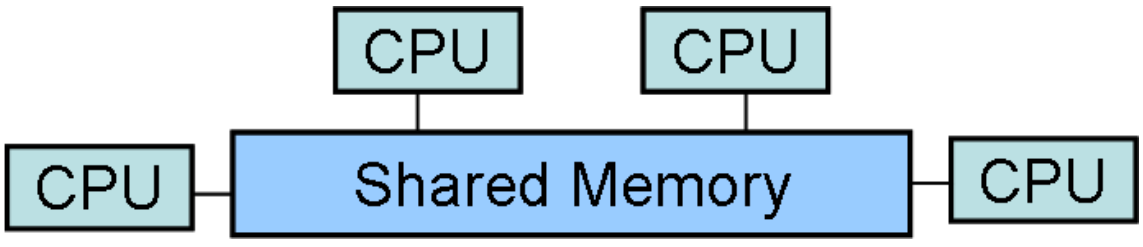

Figura 1. Memoria compartida [1]

Relativo a la memoria, este tipo de arquitectura escala proporcionalmente con el número de procesadores, ya que con cada uno se incluye una nueva región de memoria.

Sin embargo, el propio hecho de que cada procesador cuente con su propia memoria privada hace que los tiempos de acceso de un procesador a datos locales y a datos de otro procesador (a través de la red) sean heterogéneos.

Dado que en computación paralela cada proceso realiza operaciones cuyos resultados han de ser agregados o combinados, para que los procesos puedan compartir los datos procesados localmente, es necesario que exista un canal común. Este canal es generalmente la red (*ethernet*, por ejemplo).

2.1.2.- MEMORIA COMPARTIDA

A diferencia de la arquitectura de memoria distribuida, en este modelo los procesadores comparten la misma memoria pese a realizar sus tareas y operaciones independientemente. A causa de esta compartición de recursos, los datos calculados o modificados por un proceso son visibles para el resto de procesadores.

Existen dos variantes de esta arquitectura:

- UMA (*Uniform Memory Access*): También conocido como SMP (*Symmetric Multi-Processor*), todas las unidades de procesamiento son idénticas y por tanto tienen el mismo tiempo de acceso a la unidad memoria que comparten.
- NUMA (*Non-Uniform Memory Access*): En este otro caso, la arquitectura se compone de varias unidades SMP enlazadas. El acceso a memoria dentro de un multiprocesador simétrico sigue siendo uniforme, pero entre SMPs es más lento debido al pequeño *overhead* que introduce la comunicación a través del enlace.

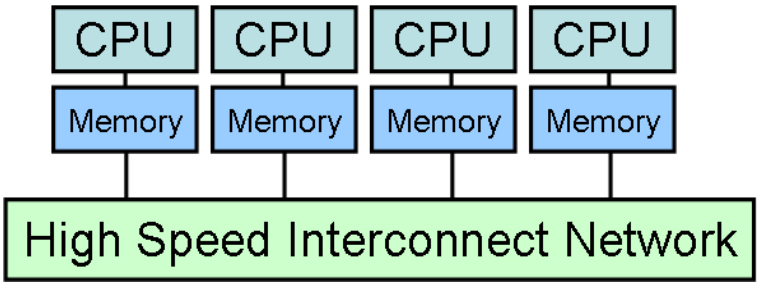

Figura 2. Memoria distribuida [1]

La principal ventaja de esta arquitectura es la disponibilidad de una memoria global y cercana a los procesadores, lo que supone un acceso a los datos bastante

uniforme y más rápido que en el caso de la memoria distribuida. A pesar de esta mejora, este modelo pierde frente al de memoria distribuida en cuanto a escalabilidad.

La inclusión de nuevas unidades de procesamiento no supone un aumento de la memoria, sino un aumento del tráfico en los buses de acceso a la memoria, incrementando el tiempo de acceso. De hecho, el tiempo de acceso a la memoria sería aún mayor, en caso de que existieran mecanismos para garantizar la coherencia de caché, debido a las constantes modificaciones sobre la misma memoria.

2.1.3.- ARQUITECTURA HÍBRIDA

Este modelo combina las dos opciones expuestas previamente. Es decir, se compone de varias unidades de memoria compartida (normalmente GPUs) que actúan independientemente. La memoria de una unidad de memoria compartida es privada para el resto de unidades.

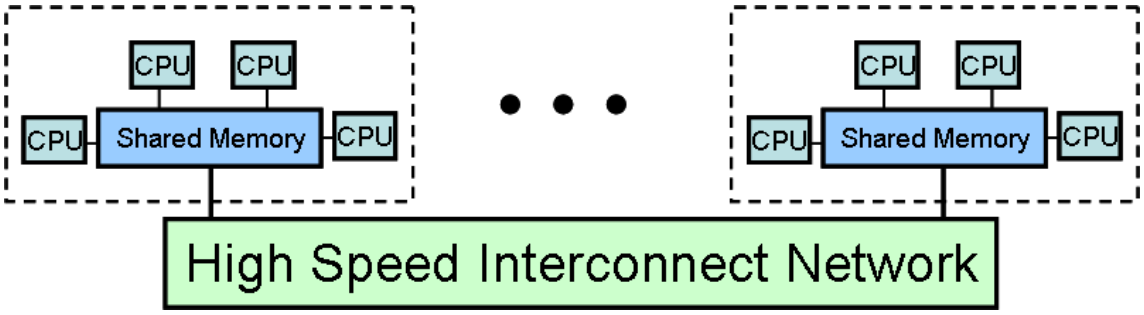

Figura 3. Arquitectura híbrida [1]

Como en el caso de la memoria distribuida, es necesario utilizar la red para intercambiar datos entre unidades, siendo, nuevamente, responsabilidad del programador especificar cómo se produce el intercambio de datos entre dichas unidades, así como la sincronización entre las mismas.

2.2- COMPUTACIÓN PARALELA BASADA EN PASO DE MENSAJES

Como se explicó previamente, en la arquitectura de memoria distribuida, cada proceso dispone de su propia memoria, la cual no es accesible para el resto de procesos. Por lo tanto, para comunicarse entre sí, los procesos intercambian datos mediante mensajes.

El intercambio de mensajes puede ser síncrono, cuando el proceso receptor debe estar preparado para recibir mensajes, o asíncrono si el emisor puede enviar mensajes con independencia de la preparación del receptor.

El paso de mensajes puede ser o no bloqueante, dependiendo de si la ejecución de la rutina en los procesos se detiene si el mensaje ha sido recibido. Por lo tanto, el paso de mensajes puede utilizarse como un mecanismo de sincronización.

El paso de mensajes es, quizás, el modelo de programación paralela más extendido.

2.3.- OTROS MODELOS DE COMPUTACIÓN PARALELA

En este apartado se presentarán modelos de computación paralela diferentes del modelo basado en paso de mensajes.

2.3.1.- PARALELISMO A NIVEL DE DATOS

Lo que propone este modelo es explotar la concurrencia dividiendo y distribuyendo los datos a tratar, de manera que procesos independientes apliquen la misma serie de operaciones sobre datos distintos.

En este modelo, no se suelen dar circunstancias en las que haya que manejar la integridad de los datos en memoria, ya que los procesos operan normalmente sobre subconjuntos disjuntos de datos. Para ello es necesario, de hecho forma parte del diseño de los programas, disponer de una buena división y asignación de datos a procesos.

Es esta necesidad de conseguir una distribución apropiada de los datos entre los procesadores, y la dificultad que ello conlleva, la causa de que el uso este modelo no esté tan extendido.

2.3.2.- MEMORIA COMPARTIDA

Al igual que en el modelo de paso de mensajes, los procesos ejecutan tareas distintas, pero operan sobre la misma memoria. Por ello, es necesario controlar los accesos a la misma para garantizar la integridad de los datos a los que acceden los procesos.

A diferencia de la computación paralela basada en paso de mensajes, para comunicarse los procesos no disponen de funciones para enviarse datos. En su lugar, para comunicarse, los procesos leen y escriben datos en determinadas posiciones de la memoria que comparten [2].

2.4.- QUÉ ES MPI

MPI (*Message Passing Interface*) es una especificación que define los tipos de datos y las funciones (la interfaz) que debe implementar una librería orientada al modelo de computación paralela basado en paso de mensajes [3].

El objetivo principal de la especificación es definir una librería con las siguientes características:

- Eficiencia: Se define una configuración de manera que se puedan enviar mensajes consumiendo el mínimo número de recursos posibles, de manera que se interfiera lo mínimo posible en la ejecución del programa que hace uso de la librería.
- Portabilidad: Las librerías que se adhieran a este estándar requieren (o no) mínimas modificaciones para ser utilizadas en diferentes plataformas.
- Flexibilidad: La especificación define una librería fácilmente extensible, de manera que se puedan desarrollar implementaciones optimizadas para plataformas o arquitecturas específicas.
- Disponibilidad: Los de comunicación entre procesos son gestionados internamente por la librería, de manera transparente para el usuario.

Pese a ser concebida inicialmente pensando en arquitecturas de memoria distribuida, con el paso del tiempo MPI ha logrado uno de sus objetivos, la portabilidad, ya que puede ser utilizado en prácticamente cualquiera de las arquitecturas citadas anteriormente: de memoria compartida, de memoria distribuida o arquitecturas híbridas.

Las operaciones definidas en el estándar se pueden clasificar en las siguientes categorías:

- Comunicación: estas operaciones implementan el paso de mensajes entre procesos. Esta comunicación puede ser:
    - Unilateral (*one-sided*): cuando un proceso accede a la memoria de otro proceso sin necesidad de sincronización (mediante un send-receive [4]).
    - Punto a punto: cuando para comunicarse, los procesos han de sincronizarse.
    - Colectiva: cuando los mensajes tienen múltiples receptores (*scatter*, *broadcast*) o varios emisores y un receptor (*gather*).
- Gestión de grupos, contexto y comunicadores: estas operaciones permiten gestionar la comunicación y se utilizan para definir otras operaciones que extiendan el estándar.

  Los comunicadores son objetos que representan grupos de procesos que se comunican entre sí. Un grupo son un "conjunto ordenado de identificadores de procesos" pertenecientes a un comunicador. El contexto es una propiedad de los comunicadores que permite diferenciar unos de otros, particionando el espacio de comunicación.

- Creación y gestión de procesos: estas operaciones permiten crear procesos dinámicamente, y permiten además que dichos procesos puedan comunicarse con los existentes.
- Entrada y salida: estas operaciones permiten la creación, la lectura y la escritura de ficheros.

Actualmente, el estándar se encuentra en su versión 3.1. Pese a ello existen propuestas y mejoras para la versión 4.0, como por ejemplo:

- Soporte para mejorar la tolerancia a fallos en aplicaciones basadas en MPI.
- Extensiones para añadir soporte a modelos de programación que agregan múltiples tecnologías (híbridos).
- *Stream messaging*
- Cancelación de envíos

2.5.- IMPLEMENTACIONES DE MPI

La especificación MPI cuenta con adaptaciones (o *bindings*) de la librería para tres lenguajes de programación: C, C++ y Fortran. En estas adaptaciones se definen las funciones e interfaces específicas que se deben implementar para cada lenguaje.

Como se mencionó anteriormente, MPI es un estándar por lo que admite diferentes implementaciones. De las diversas implementaciones existentes, las más populares son:

2.5.1.- OpenMPI

OpenMPI es una implementación de la especificación MPI, llevada a cabo por académicos y empresas colaboradoras. Esta implementación ofrece concurrencia y *'thread safety*', así como tolerancia a fallos tanto de red como en los procesos. Además, soporta todo tipo de redes. Esta librería se distribuye bajo una licencia de software libre basada en BSD License [5].

2.5.2.- MPICH

MPICH es probablemente la implementación del estándar MPI más utilizada. Esta implementación tiene como principal característica la portabilidad y que puede ejecutarse en distintas arquitecturas y redes, así como en los principales sistemas operativos (Linux, Windows, Mac OS e incluso Solaris). Esta biblioteca está concebida para ser extensible de cara a implementaciones del estándar MPI derivadas de MPICH [6].

2.5.3.- Intel MPI Library

Esta implementación forma parte de la suite Intel Parallel Studio XE [6], aunque está disponible de manera independiente. Al igual que OpenMPI y MPICH implementa la versión 3.1 del estándar MPI. Incluye optimizaciones para algunos procesadores Intel y para la arquitectura *Omni-Path* [8] de la misma compañía. En lo referente a los sistemas operativos, únicamente ofrece soporte para Linux y Windows.

2.6.- HERRAMIENTAS DE PROFILING ACTUALES

En esta sección se presentan una serie de herramientas que ofrecen funcionalidades para llevar a cabo tareas de profiling, logging, debugging u optimización de aplicaciones basadas en MPI.

- Paraver: Paraver es una herramienta de visualización de trazas que forma parte de la *suite* CEPBA-Tools. Paraver es compatible con los siguientes estándares: MPI, OpenMP, pthreads, OmpSs y CUDA. Esta herramienta permite generar e importar trazas y ofrecer una representación gráfica de las métricas importadas. De esta manera se pueden comparar las ejecuciones del mismo programa con implementaciones en distintas tecnologías o ejecuciones del mismo programa con distinta carga de datos entre otras cosas. [9]

- TAU [10]: TAU (*Tuning & Analysis Utilities*) es una herramienta de *profiling*, *tracing* y análisis de aplicaciones paralelas. Además de estas herramientas,

ofrece la posibilidad de exportar trazas compatibles con Paraver y otras herramientas y visualizar los resultados obtenidos durante el análisis del rendimiento.

- Intel Parallel Studio XE: Intel dispone de una implementación de MPI optimizada para ser ejecutada en clústeres con hardware de la misma marca(¿compañía?). Dentro del paquete Intel Parallel Studio XE, junto con MPI Intel Library, Intel ofrece una herramienta, Intel Trace Analyzer and Collector. Esta herramienta permite obtener trazas de ejecución de los programas que hacen uso de MPI. De este modo se pueden detectar cuellos de botella en las aplicaciones, analizar el rendimiento en determinadas partes de las aplicaciones, entre otras cosas.

- mpiP: Mientras que las herramientas mencionadas previamente generan trazas de toda la ejecución del programa, mpiP únicamente recoge estadísticas de las funciones MPI. Es por esto por lo que es la herramienta más parecida a la biblioteca desarrollada para este Proyecto de Fin de Grado. En mpiP cada proceso registra las métricas asociadas a llamadas a funciones de MPI, y al final de la ejecución se agregan los datos recogidos por los diferentes procesos. [11]

- MPE: MPE (*MPI Parallel Environment*) es una suite que incluye diferentes librerías y herramientas (*tracing*, *debugging* y visualización, entre otras utilidades) para facilitar el análisis de aplicaciones basadas en MPI. [12]

*IPS (Intel Parallel Studio)

	Paraver	**TAU**	**IPS XE**	**mpiP**	**MPE**
Representación gráfica	✓	✓ (Paraprof)	✓ (Trace Analyzer)	✗	✓
Generación de trazas de ejecución	✓ (Extrae)	✓	✓	✗	✓
Optimización (sugerencias)	✗	✗	✓ (VTune Amplifier [13])	✗	✗
Captura (*logging*) de funciones MPI	✗	✗	✓	✓	✓

Tabla 1. Comparativa de herramientas para MPI

3.- ANÁLISIS

En este capítulo se describe la herramienta desarrollada y se enuncian los requisitos que debe satisfacer la misma.

3.1.- DESCRIPCIÓN

Para llevar a cabo el análisis de aplicaciones basadas en MPI, se precisa de un sistema que permita capturar las llamadas a determinadas funciones del estándar.

Las funciones cuya ejecución se quiere monitorizar, MPI_Send y MPI_Recv, son las funciones en las que se basan la gran mayoría de programas que hacen uso de alguna implementación del estándar MPI.

Algunas de las herramientas mencionadas previamente permiten generar trazas de ejecución e incluso visualización de las mismas. El principal inconveniente es que su *scope* es más amplio del requerido, es decir, ofrecen más funcionalidades de las requeridas.

Por otra parte, para el análisis se busca relacionar los eventos referentes a MPI registrados durante la ejecución de los programas con el estado de la máquina durante la misma ejecución. Las métricas registradas para analizar el estado de cada nodo son el porcentaje de utilización del procesador, la temperatura, y el uso de memoria.

Por lo tanto, se decide implementar una biblioteca que permita registrar los eventos de *send* y *receive* de MPI, y enviar los datos recogidos a un servidor. Por otro lado, se extiende una aplicación servicio (o *daemon*) existente para que envíe los datos del estado de cada nodo al servidor recientemente mencionado. Toda esta información se almacena en una base de datos.

3.2.- REQUISITOS

A continuación se muestra la lista de requisitos funcionales, no funcionales y de rendimiento. Cada requisito se muestra en una tabla con la siguiente información:

- Descripción: definición de las condiciones que debe satisfacer el sistema a implementar.
- Identificador: código que identifica el requisito.
- Justificación: justificación de la importancia del requisito.
- Prioridad: en este campo se indica el grado de importancia del requisito.
- Verificabilidad: grado en el que se puede comprobar el cumplimiento del requisito.

3.2.1.- REQUISITOS FUNCIONALES

Identificador	RF-01	**Prioridad**	Alta	**Verificabilidad**	Alta
Descripción	La biblioteca debe capturar las llamadas a la función MPI_Send.				
Justificación	MPI_Send es una de las funciones más utilizadas en MPI y la función en la que se basan internamente algunas otras funciones. Para poder analizar el paso de mensajes, es necesario saber cuándo éstos son enviados.				

Tabla 2. Requisito funcional 1

Identificador	RF-02	**Prioridad**	Alta	**Verificabilidad**	Alta
Descripción	La biblioteca debe capturar las llamadas a la función MPI_Recv.				
Justificación	Al igual que MPI_Send, MPI_Recv es una función esencial en MPI. Dado que generalmente existe sincronización entre proceso emisor y el proceso receptor, es necesario saber cuándo se produce la recepción de mensajes.				

Tabla 3. Requisito funcional 2

Identificador	RF-03	**Prioridad**	Alta	**Verificabilidad**	Alta
Descripción	Por cada llamada a cualquiera de las funciones citadas en los requisitos anteriores, la biblioteca debe generar un mensaje con la información relativa a la invocación de la función correspondiente.				
Justificación	Capturando la información referente a la invocación de funciones MPI, se puede conocer cómo se desarrolla la ejecución de las aplicaciones. Agrupar dicha información en un mensaje por invocación de función permite trabajar las métricas capturadas con mayor facilidad.				

Tabla 4. Requisito funcional 3

Identificador	RF-04	**Prioridad**	Alta	**Verificabilidad**	Alta
Descripción	El mensaje propuesto en el requisito RF-03, debe contener un campo en el que se especifique la función invocada.				
Justificación	Únicamente se van a capturar 2 funciones (MPI_Send y MPI_Recv), pero es interesante para el análisis saber cuál es la que se invoca en cada instante.				

Tabla 5. Requisito funcional 4

Identificador	RF-05	**Prioridad**	Alta	**Verificabilidad**	Alta
Descripción	En el mensaje de monitorización de funciones MPI, se deben mostrar información identificativa de los procesos involucrados (emisor y receptor).				
Justificación	Con la identificación de procesos se pueden descubrir patrones de comunicación entre procesos, se puede obtener el proceso con mayor actividad en cuanto a paso de mensajes, entre otras métricas y observaciones.				

Tabla 6. Requisito funcional 5

Identificador	RF-06	**Prioridad**	Alta	**Verificabilidad**	Alta
Descripción	El mensaje de monitorización de funciones MPI debe incluir el tamaño del mensaje enviado o recibido en cada caso.				
Justificación	El registro del tamaño de los mensajes enviados es útil para la detección de patrones durante la ejecución de aplicaciones basadas en MPI.				

Tabla 7. Requisito funcional 6

Identificador	RF-07	**Prioridad**	Alta	**Verificabilidad**	Alta
Descripción	Se enviará cada mensaje de monitorización MPI a un servidor remoto.				
Justificación	Puesto que los procesos pueden estar en distintos nodos, es necesario que exista un servidor que recoja todos los datos de monitorización enviados desde cada proceso.				

Tabla 8. Requisito funcional 7

Identificador	RF-08	**Prioridad**	Alta	**Verificabilidad**	Alta
Descripción	Se deberá almacenar cada mensaje monitorización recibido (del formato definido en el requisito RF-02).				
Justificación	Para realizar el análisis y generar gráficas es necesario persistir los datos.				

Tabla 9. Requisito funcional 8

Identificador	RF-09	**Prioridad**	Alta	**Verificabilidad**	Alta
Descripción	Por cada mensaje de monitorización MPI persistido, y junto a los datos contenidos en el mensaje, se deberá almacenar un *timestamp* del momento en el que fue recibido por el servidor de monitorización MPI.				
Justificación	Incluir el momento de recepción del mensaje permite poner las métricas y datos recogidos en contexto de cara a llevar a cabo un análisis más completo.				

Tabla 10. Requisito funcional 9

Identificador	RF-10	**Prioridad**	Media	**Verificabilidad**	Alta
Descripción	Periódicamente se generará un mensaje de monitorización de cada nodo en el que se ejecuten procesos de la aplicación basada en MPI con los siguientes datos: - Porcentaje de utilización del procesador - Porcentaje de uso de la memoria - Temperatura				
Justificación	Las métricas sugeridas pueden ser útiles para descubrir si existe correlación (y si la hay, en qué medida) entre el tráfico y el uso de memoria o la temperatura, por ejemplo.				

Tabla 11. Requisito funcional 10

Identificador	RF-11	**Prioridad**	Media	**Verificabilidad**	Alta
Descripción	Se enviará cada mensaje de monitorización del *daemon* al mismo servidor remoto utilizado.				
Justificación	Puesto que los procesos pueden estar en distintas máquinas, es necesario que exista un servidor que recoja todos los datos de monitorización enviados desde cada nodo o máquina que ejecute procesos de la aplicación basada en MPI. Además, enviando toda la información de monitorización al mismo servidor es más fácil procesarla, sea información relativas a funciones MPI como del estado de los nodos.				

Tabla 12. Requisito funcional 11

Identificador	RF-12	**Prioridad**	Media	**Verificabilidad**	Alta
Descripción	Por cada mensaje de monitorización de los nodos se almacenará, junto con el contenido de cada mensaje, el momento (*timestamp*) en el que fue recibido por el servidor remoto de monitorización MPI.				
Justificación	Incluir el momento de recepción de los mensajes de monitorización del *daemon* permite representar las métricas capturadas en función del tiempo. Por otra parte, se podrá utilizar este dato para relacionarlos con los datos obtenidos durante la monitorización de funciones MPI.				

Tabla 13. Requisito funcional 12

Identificador	RF-13	**Prioridad**	Alta	**Verificabilidad**	Alta
Descripción	Los datos de monitorización relacionados con MPI se deben almacenar en un lugar distinto al lugar en el que se almacenan los datos de monitorización de los nodos.				
Justificación	Separar el almacenamiento de mensajes distintos ofrece flexibilidad a la hora de seleccionar los datos de monitorización que se quieran emplear para el análisis.				

Tabla 14. Requisito funcional 13

3.2.2.- REQUISITOS NO FUNCIONALES

En las siguientes tablas, que tienen la misma estructura que las tablas de los requisitos funcionales, se muestran aquellos requisitos relativos al rendimiento del sistema desarrollado.

Identificador	RNF-01	**Prioridad**	Alta	**Verificabilidad**	Media
Descripción	El envío de datos de monitorización tanto de funciones MPI como del estado de los nodos deberá lo más rápido posible.				
Justificación	El proceso de monitorización debe introducir el mínimo *overhead* posible, para reducir su influencia en las métricas registradas.				

Tabla 15. Requisito no funcional 1

Identificador	RNF-02	**Prioridad**	Alta	**Verificabilidad**	Media
Descripción	El servidor de monitorización deberá ser concurrente.				
Justificación	Por lo general habrá más de un proceso en ejecución simultánea, por eso el servidor deberá ser capaz de recibir los mensajes que dichos procesos envíen.				

Tabla 16. Requisito no funcional 2

Identificador	RNF-03	**Prioridad**	Media	**Verificabilidad**	Alta
Descripción	El usuario podrá configurar la dirección IP del servidor remoto de monitorización, así como los puertos de escucha de mensajes de monitorización.				
Justificación	Incluir el momento de recepción de los mensajes de monitorización GPU permite representar las métricas capturadas en función del tiempo. Por otra parte, se podrá utilizar este dato para relacionarlos con los datos obtenidos durante la monitorización de funciones MPI.				

Tabla 17. Requisito no funcional 3

4.- DISEÑO E IMPLEMENTACIÓN

En este capítulo se abordan los procesos de diseño e implementación, incluyendo justificaciones de las decisiones tomadas, así como la exposición de las diferentes alternativas contempladas.

El sistema diseñado consta de tres partes claramente diferenciadas: el servidor, el servicio de monitorización de nodos y la biblioteca de monitorización de funciones MPI. El servidor de monitorización es el componente que se encarga de almacenar en la base de datos los distintos mensajes de monitorización enviados por el resto de entidades que conforman el sistema.

Si bien el *daemon* y la biblioteca de monitorización de funciones MPI son clientes del servidor de monitorización, no se agrupan en una categoría común ya que tanto la información que recogen, como la periodicidad del envío de mensajes, como su implementación es diferente.

El servidor de monitorización únicamente espera que lleguen mensajes de monitorización para persistir los datos que contienen (junto con el momento en el que los mensajes son recibidos).

Por otro lado, el servicio de monitorización en cada nodo envía periódicamente métricas relativas al estado de cada una de las unidades de procesamiento (CPUs o GPUs, si tiene) de ese nodo al servidor. La frecuencia con la que estos mensajes de monitorización se envían se puede configurar al ejecutar el *daemon*. A su vez, la biblioteca de monitorización de funciones MPI desarrollada envía mensajes de monitorización al servidor solamente cuando alguno de los procesos invoca alguna de las dos funciones a monitorizar.

En la siguiente figura se representa la arquitectura del sistema desarrollado, indicando qué componentes lo forman.

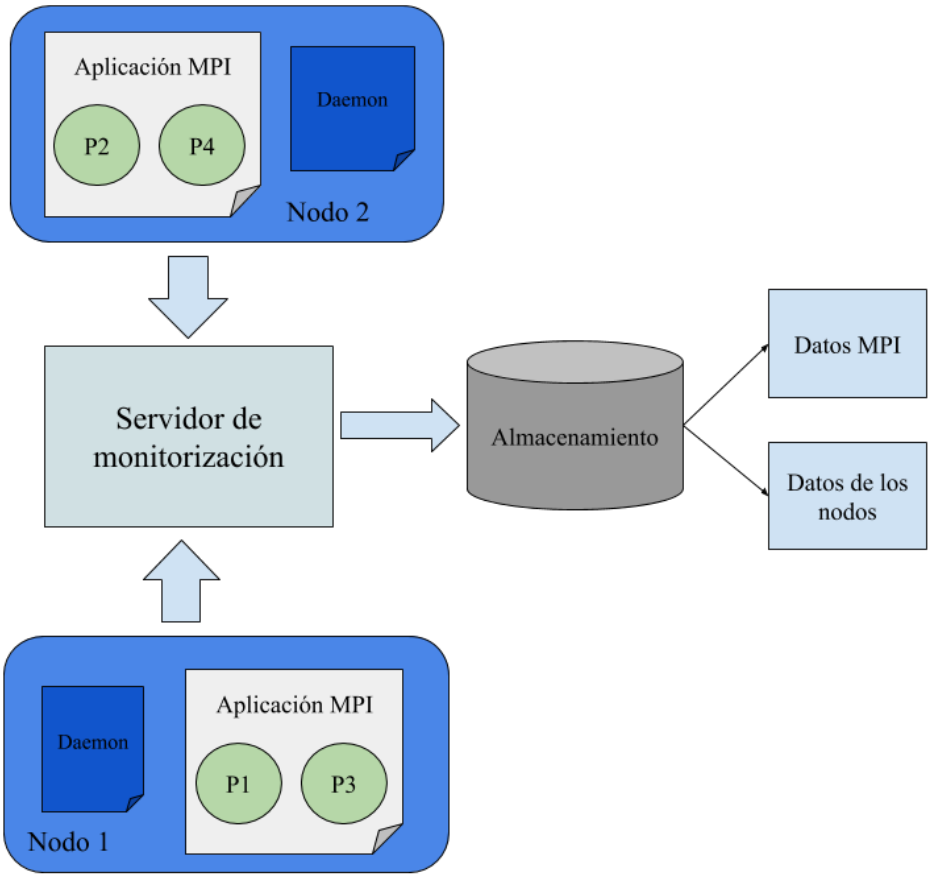

Figura 4. Arquitectura del sistema distribuido de monitorización

En el esquema se ve cómo la ejecución de la aplicación a monitorizar, concretamente la distribución de procesos, se reparte entre varios nodos (puede haber un único nodo si se desea). En cada uno de estos nodos se ejecuta el daemon para obtener las métricas relativas al estado de la máquina.

En los apartados sucesivos se procede a explicar con mayor detalle el funcionamiento y el diseño de cada uno de los tres componentes citados.

4.1.- BIBLIOTECA DE MONITORIZACIÓN MPI

La biblioteca de monitorización tiene dos cometidos principales: capturar información relacionada con las invocaciones a las funciones MPI_Send y a MPI_Recv, y enviar al servidor la información recogida.

El lenguaje de programación escogido para el desarrollo de la biblioteca es C, uno de los tres lenguajes para los que el estándar MPI tiene *binding*. Otros aspectos de C que propiciaron su elección son la portabilidad (C es *cross-platform*) y la escasa cantidad de recursos que son necesarios para su ejecución.

Para implementar la primera funcionalidad, se ha sobrescrito la implementación de MPICH de los dos métodos a monitorizar. Para sobrescribir la implementación de las funciones MPI_Send y MPI_Recv se ha hecho uso de la interfaz de *profiling* PMPI (Profiling Message-Passing Interface).

Dado que la interfaz de *profiling* ofrece las mismas funcionalidades que la interfaz estándar, es equivalente llamar a la función MPI_Send que a la función PMPI_Send. Esto permite sobrescribir la función MPI_Send, obtener y enviar los datos al servidor remoto e llamar internamente a la función PMPI_Send de manera que este proceso de monitorización sea transparente a la aplicación en ejecución.

Durante la reimplementación de las funciones a monitorizar, se crearon dos ficheros, "send.c" y "recv.c", que contienen la nueva implementación de MPI_Send y MPI_Recv respectivamente. Para enviar el mensaje con los datos de monitorización ambas implementaciones hacen uso de la función "send_mpi_monitoring_info" definida en el fichero "mpi_monitor.h" e implementada en el fichero "mpi_monitor.c".

Todas estas clases se compilan para generar una biblioteca estática. Ésta será enlazada con la aplicación basada en MPI a monitorizar, para sobrescribir la implementación MPI que use por defecto dicha aplicación.

El comando para generar dicha biblioteca es:

ar rcs lib<nombre de la biblioteca>.a send.o recv.o mpi_monitor.o

Tabla 18. Compilación de la biblioteca

El comando para compilar la aplicación incluyendo la biblioteca desarrollada es:

mpicc <nombre del código fuente de la aplicación> -o <nombre de la aplicación> -l <nombre de la biblioteca generada> -L.

Tabla 19. Compilación de la aplicación basada en MPI

* Previamente, cada una de las tres clases que forman la biblioteca deben ser compiladas individualmente con las bibliotecas de MPI (*lmpi* y *lpmpi*)

mpi_monitor.h

Este fichero incluye la definición del método ***send_mpi_monitoring_info***, utilizado para enviar los datos obtenidos al servidor de monitorización. Además,

contiene la definición de la estructura de datos (*MPI_Monitor_Info*) a enviar mediante la función mencionada previamente. A continuación se incluye dicha definición:

```
typedef struct mpi_monitor_info{
  char operation;
  short source;
  short dest;
  long msg_size;
} MPI_Monitor_Info;
```

Tabla 20. Estructura del mensaje de monitorización MPI

Lo que se pretende con esta estructura es minimizar el tamaño del mensaje a enviar, de manera que eso se traduzca en mayor velocidad de ejecución de la función ***send_mpi_monitoring***, y por consiguiente en menor *overhead*. Para ello, para cada atributo o campo de la estructura, se va a proponer el tipo de datos más pequeño que permita representar el rango de valores posibles considerado para dicho campo.

El atributo "*operation*" permite identificar la función MPI que se invoca en cada caso. Únicamente se van a monitorizar 2 funciones por lo que con un char se pueden codificar las 2 opciones (1 char = 1 byte = 8 bits = 2^8 = 256 valores representables). Además, para este campo la codificación de las operaciones es la siguiente: 's' = MPI_Send, 'r' = MPI_Recv.

Se proponen dos campos, "*source*" y "*dest*", para identificar al emisor y al receptor respectivamente. El tipo de dato para cada uno de estos campos es el short, permitiendo poder representar hasta 65.536 procesos, cifra que dudosamente se alcanzará (1 short = 2 bytes = 16 bits = 2^{16}= 65.536 valores).

Para representar la longitud del mensaje enviado o recibido, se incluye el campo "*msg_size*". Puesto que a priori no se sabe qué tipo de datos ni qué cantidad de ellos se envían, este campo es de tipo long, de manera que se pueda representar cualquier tamaño de mensaje (1 long = 4 bytes = 32 bits = 2^{32}= 4.294.967.296 valores).

mpi_monitor.c

Este fichero contiene la implementación de la función que envía el mensaje definido en el requisito funcional RF-02, ***send_mpi_monitoring_info***, al servidor de monitorización.

Dicha función genera el un mensaje con el formato requerido al servidor mediante un datagrama UDP. Se optó por utilizar UDP dado que no es orientado a conexión como TCP, y por lo tanto es rápido como se indica en los requisitos. Otro motivo por el que finalmente se escogió UDP es que en caso de que existan errores al enviar el mensaje, TCP realiza reintentos, lo que puede ralentizar la ejecución normal de la función invocada (MPI_Send o MPI_Recv).

send.c

De acuerdo con el estándar MPI, la función MPI_Send debe recibir los siguientes parámetros:

- La posición inicial de buffer donde se encuentran los datos a enviar.
- El número de datos a enviar.
- El tipo de datos a enviar.
- El *rank* o identificador del proceso receptor.
- Una etiqueta de mensaje.
- El comunicador que contiene los procesos que intercambian mensajes.

Con estos parámetros se dispone de la información necesaria para satisfacer los requisitos definidos relativos a los mensajes de monitorización de MPI, excepto el identificador del proceso emisor. Para obtener este último atributo, el identificador del proceso emisor, se hizo uso de la función MPI_Comm_rank. Esta función devuelve el identificador o rango del proceso que la invoca, en este caso el emisor.

Una vez capturados todos los datos requeridos, se envían en un mensaje con el formato especificado en el requisito funcional RF-02, usando la función ***send_mpi_monitoring_info***.

Tras enviar el mensaje de monitorización, se llama a la función PMPI_Send con los mismos parámetros y el mismo orden en que fueron recibidos por la nueva implementación MPI_Send. Invocar la función PMPI_Send es equivalente a llamar a la implementación original de MPI_Send.

```
int MPI_Send(const void *buf, int count, MPI_Datatype
datatype, int dest, int tag, MPI_Comm comm)
{
    int sender; // ID (rank) of the sender process
    int datatype_size;
    MPI_Type_size(datatype, &datatype_size);
    MPI_Comm_rank(comm, &sender);

    send_mpi_monitoring_info(dest, sender, (long)(count *
datatype_size), 's');

    return PMPI_Send(buf, count, datatype, dest, tag, comm);
}
```

Tabla 21. Implementación de la función MPI_Send

recv.c

Según el estándar MPI, la función MPI_Recv debe recibir los siguientes parámetros:

- La posición inicial de buffer donde se almacenan los datos recibidos
- El número máximo de datos a recibir
- El tipo de datos a recibir
- El *rank* o identificador del proceso emisor
- Una etiqueta de mensaje
- El comunicador que contiene los procesos que intercambian mensajes

Como en el caso de MPI_Send, se dispone de todos los datos a excepción del identificador del proceso receptor. De nuevo se procedió a utilizar la función MPI_Comm_rank para obtener el rango del proceso receptor.

Todos los datos capturados son enviados al servidor de monitorización usando la función ***send_mpi_monitoring_info***.

Después de haber enviado los datos al servidor, se llama a la función PMPI_Recv para la nueva implementación finalice con el comportamiento original de la función MPI_Recv.

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int
source, int tag, MPI_Comm comm, MPI_Status *status)
{
    int receiver; // ID (rank) of the current (receiver)
process
    int datatype_size;
    MPI_Type_size(datatype, &datatype_size);
    MPI_Comm_rank(comm, &receiver);

    send_mpi_monitoring_info(receiver, source, (long)(count *
datatype_size), 'r');

    return PMPI_Recv(buf, count, datatype, source, tag, comm,
status);
}
```

Tabla 22. Implementación de la función MPI_Recv

4.2.- SERVIDOR DE MONITORIZACIÓN

El servidor de monitorización es el componente del sistema encargado de almacenar en la base de datos todos los datos de monitorización recogidos.

Pese a que internamente se compone de dos servidores, uno para cada tipo de cliente, ambos servidores se ejecutan como parte de la misma aplicación.

Al igual que la librería que captura los datos relativos a MPI está escrito en C, en busca de la homogeneidad del sistema y de que los requisitos de configuración de cada componente, y los del sistema en general, sean los mínimos posibles.

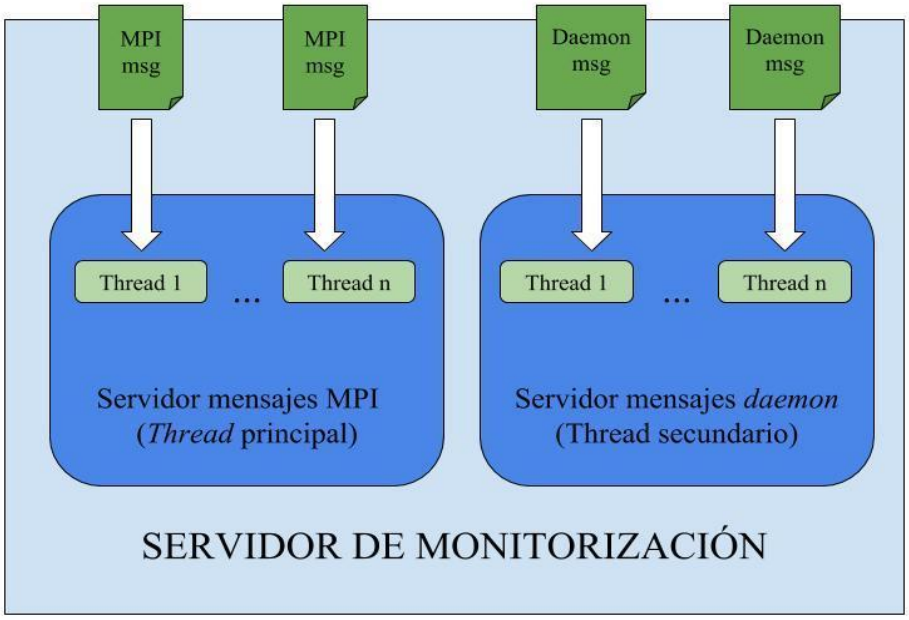

Figura 5. Servidor de monitorización

4.2.1.- ALMACENAMIENTO

Para poder comparar y analizar el rendimiento entre diferentes ejecuciones, es necesario persistir la información de monitorización obtenida. Es por esto por lo que el almacenamiento es un componente esencial del sistema de monitorización.

Durante el diseño y la implementación del sistema se ha tratado de ajustar el número y el alcance de las funcionalidades ofrecidas por el sistema, cumpliendo los requisitos obtenidos durante el análisis.

Los objetivos en cuanto al almacenamiento son: velocidad, portabilidad, mínima configuración y, sobre todo, la facilidad para tratar los datos almacenados, ya que cabe recordar que la finalidad del sistema facilitar el análisis del rendimiento de las aplicaciones. Para intentar alcanzar dichos objetivos, algunas de las opciones barajadas fueron: fichero de texto plano, fichero JSON y SQLite.

A continuación se muestra una comparativa entre las diferentes opciones contempladas, indicando el grado en que satisfacen los requisitos de rendimiento referentes al almacenamiento. Por cada atributo del componente de almacenamiento se ordenan las opciones en orden ascendentemente de idoneidad (la mejor opción tiene el valor más alto)

	Fichero plano	**JSON**	**SQLite**
Necesidad de configuración	3 (las 3 opciones son prácticamente *plug and play*)		
Portabilidad	2	2	3
Facilidad para manejar datos almacenados	1	2	3
Velocidad	2	2	3

Tabla 23. Comparativa de las opciones de almacenamiento barajadas

En cuanto a la necesidad de configuración, las diferentes opciones tienen la máxima puntuación ya que únicamente es necesario definir la estructura con la que se almacenará la información.

En el resto de apartados, las diferentes opciones tienen puntuaciones similares, excepto en el apartado relativo al manejo de datos almacenados. Éste último ha sido el factor determinante a la hora de escoger el tipo de almacenamiento.

Si los datos almacenados fueran simplemente un registro histórico de las operaciones realizadas, cualquier opción (quizás el fichero plano es la más simple) hubiera servido. Pero para realizar el análisis SQLite ofrece mayor flexibilidad con funcionalidades como: realizar consultas, agrupar y realizar múltiples operaciones sobre los datos almacenados.

SQLite

Para persistir los datos de monitorización se ha optado por la librería SQLite, actualmente "*el motor de base de datos más utilizado del mundo*" [14].

Figura 6. SQLite

Algunos de los motivos por los cuales SQLite ha sido la opción escogida son:

- Uso extendido de la librería: esta característica se traduce en mayor soporte, lo que favorece la portabilidad en caso de que se desee utilizar el sistema desarrollado en múltiples plataformas.
- Compacidad: Todo el código fuente de la librería está contenido en un único fichero. Además, la implementación de SQLite tiene un tamaño reducido, que en la mayoría de casos no alcanza el MiB. Por otra parte, la base de datos entera (tablas, vistas, etcétera) está contenida en un único fichero, lo que minimiza el número de artefactos en el sistema.
- Completitud: SQLite ofrece todas las funcionalidades y operaciones propias de SQL, las cuales son muy útiles para relacionar todos los datos recogidos.
- Portabilidad: El archivo que contiene la base de datos es *cross-platform*, es decir, se puede importar en cualquier sistema, lo que supone otra mejora en lo referente a la portabilidad del sistema.
- Velocidad: el acceso a datos (lectura y escritura) utilizando SQLite es por lo general y de acuerdo con los desarrolladores de la librería, "*más rápido que el acceso a archivos mediante el sistema de ficheros*". Uno de los requisitos del sistema es la rapidez, por lo tanto, era necesario disponer de un mecanismo de almacenamiento que introdujese el mínimo retardo posible.

Se ha creado un fichero db_handler.c que contiene las implementaciones de los métodos necesarios para el manejo de la base de datos. Éstos son:

create_table

Esta función, si no existen, crea las tablas utilizadas para almacenar los distintos tipos de mensajes, "MPI_LOG" y "DAEMON_LOG" respectivamente.

log_daemon_info

Este método almacena los campos del mensaje de monitorización de nodos recibido, en la tabla "DAEMON_LOG".

log_mpi_info

Este método almacena los valores incluidos en el mensaje de monitorización MPI recibido, en la tabla “MPI_LOG”.

4.2.2.- FUNCIONAMIENTO

El servidor en su función principal, la función *main*, inicializa la base de datos e inicializa todos los elementos necesarios para poder atender peticiones de los clientes: la librería de monitorización MPI y el servicio de monitorización de los nodos. Las funciones que conforman el servidor se explicarán con más detalle en los apartados sucesivos.

main

Primero se recogen los argumentos requeridos para ejecutar el servidor: los puertos en los que se escuchará para recibir mensajes de monitorización de MPI y las GPUs respectivamente.

A continuación, se comprueba la validez de los puertos elegidos, y caso de cumplir los requisitos se inicializan los sockets a través de los cuales se reciben los de mensajes de monitorización. En caso contrario se termina la ejecución del servidor.

Por último, el proceso principal crea un hilo que atenderá las peticiones con datos del *daemon*. Puesto que los mensajes que reciben los 2 hilos son distintos y se procesan de manera independiente, el *thread* que gestiona los mensajes enviados por el servicio se crea como *detached*. De este modo, se pueden seguir recibiendo mensajes de monitorización de MPI si la ejecución del *daemon* falla.

init_socket

Este método inicializa los sockets de tipo UDP que reciben los mensajes con los datos de monitorización. Esta función es invocada dos veces, una por cada tipo de mensaje, pasando como argumentos el puerto especificado para cada tipo de mensaje, así como la variable donde se almacenará el identificador asignado a cada socket.

manage_mpi_info

Esta función implementa el servidor que recibe los mensajes con información de las llamadas a funciones MPI. Esencialmente se trata de un bucle infinito en el que se esperan datagramas entrantes con datos de MPI. Por cada datagrama recibido se crea un hilo encargado de almacenar el mensaje recibido. Para insertar el mensaje recibido en la base de datos, esta función llama al método *log_mpi_info*.

manage_daemon_info

Esta función idéntica a la anterior con la salvedad de que ésta gestiona datagramas con datos de monitorización de los nodos, en lugar de MPI. Una vez recibido el mensaje desde el servicio de monitorización la función invoca la función *log_daemon_info*.

4.3.- DAEMON

El servicio de monitorización, o *daemon*, es una aplicación que se ejecuta en cada nodo que participe en la ejecución paralela de la aplicación basada en MPI a analizar.

Esta aplicación recoge distintas métricas acerca del estado de la máquina, como: la temperatura, el porcentaje de utilización de la CPUs, la dirección IP del nodo, el porcentaje de memoria dinámica utilizada o el número de CPUs por nodo, entre otras.

Este servicio fue desarrollado con anterioridad a la realización de este proyecto. Por lo tanto, incorpora funcionalidades y un diseño que han debido ser adaptados para obtener los datos necesarios para elaborar el análisis de las aplicaciones basadas en MPI.

4.3.1.- FUNCIONAMIENTO

El servicio de monitorización procesa diversos ficheros para obtener información actualizada acerca del estado del sistema. En la siguiente tabla se muestran los datos de estado del sistema junto con su origen o el mecanismo mediante el cual se extraen.

Dato	Origen
Información de CPU (porcentaje de utilización)	/proc/stat

Dispositivos	/proc/diskstats
GPUs (temperatura, consumo de energía, etcétera)	Librerías de CUDA
Memoria	/proc/meminfo
Interfaces de red	/proc/class/net
Temperatura del sistema	/sys/bus/platform/devices
Consumo de energía del sistema (julios)	/sys/devices/virtual/powercap/intel-rapl

Tabla 24. Fuentes de los datos de monitorización del daemon

Los datos obtenidos en cada muestra se empaquetan hasta alcanzar el número de muestras por paquete especificado. A continuación se muestra la composición tanto de los paquetes que genera el *daemon* como de las muestras que forman dichos paquetes.

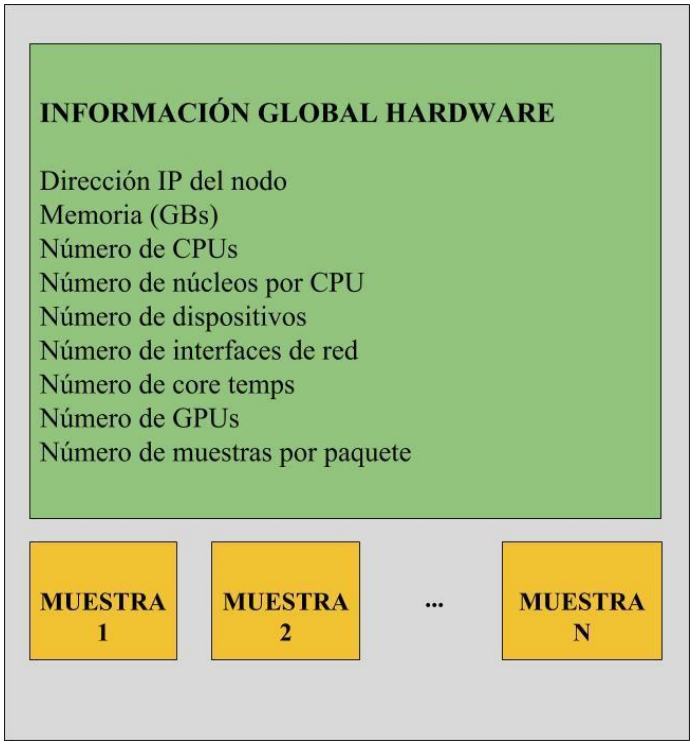

Figura 7. Paquete de monitorización generado por el daemon

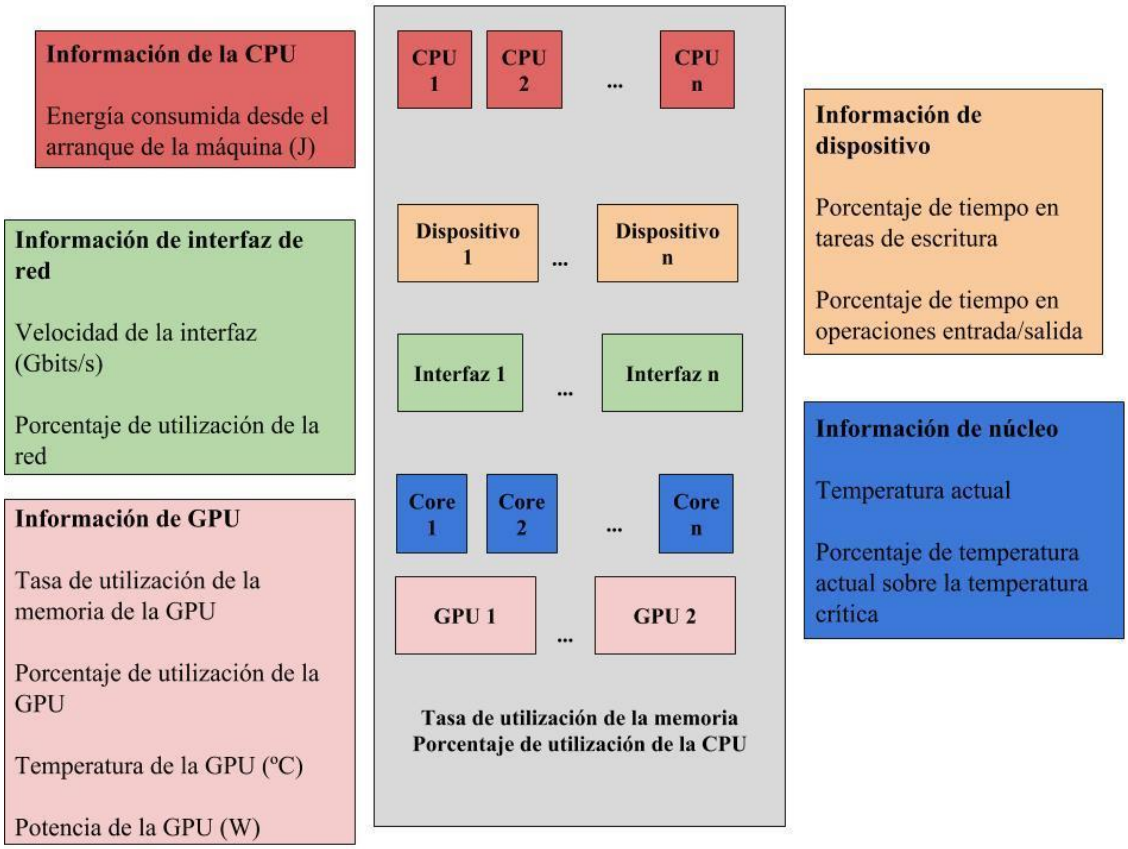

Figura 8. Muestra generada por el daemon

En cuanto al flujo de ejecución del *daemon*, la aplicación genera los paquetes de monitorización en base a 2 parámetros: el número de muestras por paquete y la frecuencia o intervalo de muestreo, ambos configurables al iniciar la aplicación. El *daemon* envía paquetes cada vez que éstos están completos, es decir, cuando contienen el número de muestras especificado.

Para ofrecer mayor cantidad de datos, en la versión extendida del *daemon*, los mensajes de monitorización (no los paquetes definidos en la figura anterior) se envían cada vez que se genera una muestra, es decir, n milisegundos, donde n es la frecuencia de muestreo. En la siguiente figura se representa el flujo de ejecución del *daemon*.

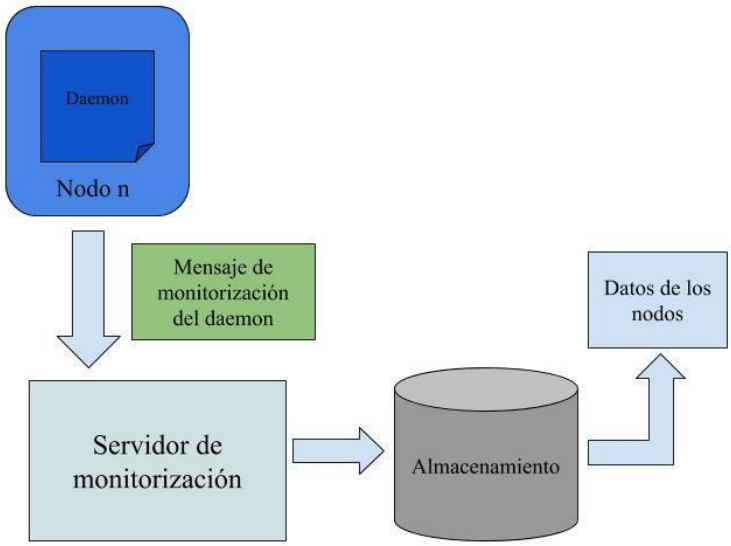

Figura 9. Flujo de ejecución del daemon

4.3.2.- EXTENSIÓN DEL SERVICIO

Para enviar al servidor de monitorización los datos especificados en los requisitos funcionales, no se modificado ninguna funcionalidad ofrecida por defecto por el *daemon*.

Lo que se ha hecho para cumplir con los requisitos ha sido añadir una nueva función a la clase "Packed_sample.c". Esta clase se encarga de agrupar todos los atributos del sistema, así como las diferentes métricas especificadas en los esquemas anteriores. El método añadido, "send_node_monitoring_info", tiene un funcionamiento al método "send_mpi_monitoring_info", definido en la librería implementada.

También se ha modificado el *header file* "Packed_sample.h" para incluir la definición de la estructura de datos que el mensaje definido en los requisitos funcionales relativos a los datos de monitorización de los nodos. Dicha definición se muestra a continuación:

```
typedef struct node_monitoring_info{
  int memory_usage;
  int cpu_usage;
  int temperature;
} Node_Monitor_Info;
```

Tabla 25. Estructura del mensaje de monitorización del daemon

La función incluida es invocada cada vez que se empaqueta una muestra, de este modo se dispondrá de una representación precisa del estado de cada máquina durante la ejecución de aplicaciones basadas en MPI.

send_node_monitoring_info

Esta función recibe como parámetros la temperatura, la tasa de utilización de la memoria y el porcentaje de utilización del procesador, para cada CPU o para cada GPU (definir). Con los parámetros recibidos genera una estructura de datos que se ajusta al tipo de mensaje definido en el requisito RF-09.

La estructura generada con los datos de monitorización de la máquina se envía al servidor haciendo uso del protocolo UDP. Nuevamente UDP es el protocolo escogido porque, a diferencia de TCP, no requiere establecer una conexión previa al envío de datos. Por otro lado, para el análisis del rendimiento de las aplicaciones se pretende disponer de muchas muestras, por lo que la frecuencia de muestreo será alta y la pérdida de datagramas no será crítica.

4.3.3.- UTILIZACIÓN

Para ejecutar esta aplicación es necesario especificar los siguientes argumentos:

./DaeMon -i <interval> -s <size> -t <threshold> -p <port> -n <port> -a <address >

Tabla 26. Comando de ejecución del daemon

Donde:

- i (*interval*): es el intervalo muestro o la frecuencia con la cual se toman muestras medida en milisegundos.
- s (*size*): es el tamaño de los paquetes, o el número de muestras por paquete, siendo 1 el tamaño mínimo y 255 el máximo.
- t (*threshold*): es el umbral [0, 100].
- n: es puerto en el que el servidor de monitorización del daemon desarrollado para el sistema de monitorización escucha. Dicho puerto debe estar en el rango [1024, 65535].
- p (*port*): es puerto en el que el servidor del daemon escucha. Dicho puerto debe estar en el rango [1024, 65535].
- a (*address*): es la dirección IP o el nombre del servidor de monitorización.

La aplicación incluye por defecto un parámetro para indicar el puerto en el que escuchara el socket del servidor (p). Dado que el tipo de mensaje que se envía al puerto

“p” es distinto al tipo de mensaje que espera recibir el servidor de monitorización, se ha decido añadir otro parámetro para configurar un puerto alternativo (n) al que enviar datos de los nodos requeridos para este proyecto.

5.- VERIFICACIÓN Y EVALUACIÓN DEL SISTEMA

En este capítulo se describen las pruebas diseñadas para verificar el cumplimiento de los requisitos funcionales extraídos durante la fase de análisis. Se citarán también los materiales y sus respectivas configuraciones, tanto hardware como aplicaciones, utilizados para implementar las pruebas.

5.1.- PLAN DE PRUEBAS

A continuación se enumeran las pruebas realizadas para comprobar el correcto funcionamiento del sistema de acuerdo con los requisitos funcionales.

Para comprobar la correcta implementación de las funcionalidades requeridas se ha hecho uso de la aplicación “ping-pong”. Se considera que la aplicación “ping-pong” es idónea para probar el sistema desarrollado ya que únicamente hace uso de las funciones MPI requeridas. En cuanto al funcionamiento, la aplicación consiste en 2 procesos que se envían un número entero, incrementando su valor en una unidad cada vez que lo reciben del otro proceso.

El flujo de ejecución de la aplicación permite verificar la recepción correcta de los mensajes de monitorización, ya que el patrón de mensajes enviados y recibidos es característico.

Cada prueba se detalla usando una tabla que contiene los siguientes campos:

- Identificador: código que permite distinguir unas pruebas de otras y que se utilizará en la matriz de trazabilidad para relacionar los requisitos funcionales con las pruebas que los cubren
- Objetivo: es la finalidad de la prueba, concretamente, el componente o caso de uso del sistema desarrollado que se quiere comprobar.
- Descripción: procedimiento seguido para alcanzar el objetivo especificado en el campo anterior.

Identificador	PR-01

Objetivo	Verificar que únicamente se capturan invocaciones a las funciones MPI_Send y MPI_Recv.
Descripción	Se crea una aplicación, que únicamente llama a la función MPI_Init. Se compila con la biblioteca desarrollada y se ejecuta.
Resultado esperado	El servidor de monitorización no debe recibir ningún mensaje de monitorización relativo a MPI.
Resultado obtenido	Satisfactorio

Tabla 27. Prueba 1

Identificador	PR-02
Objetivo	Comprobar que se obtienen correctamente los datos requeridos para la función MPI_Send.
Descripción	Se incluye en la biblioteca la impresión condicional de los datos recogidos. (Imprime los datos si la variable de entorno DEBUG_SEND vale 1) Se compila con la biblioteca desarrollada y se ejecuta con la variable de entorno DEBUG_SEND puesta a 1.
Resultado esperado	Impresión de los 4 datos requeridos para la función MPI_Send.
Resultado obtenido	Satisfactorio

Tabla 28. Prueba 2

Identificador	PR-03
Objetivo	Comprobar la obtención de los datos requeridos para la función MPI_Recv.
Descripción	Se incluye en la biblioteca la impresión condicional de los datos recogidos. (Imprime los datos si la variable de entorno DEBUG_RECEIVE vale 1) Se compila con la biblioteca desarrollada y se ejecuta con la variable de entorno DEBUG_RECEIVE puesta a 1.
Resultado esperado	Impresión de los 4 datos requeridos para la función MPI_Recv.
Resultado obtenido	Satisfactorio

Tabla 29. Prueba 3

Identificador	PR-04

Objetivo	Verificar el envío del mensaje de monitorización MPI al servidor.
Descripción	Se modifica el servidor para imprimir el contenido del mensaje de monitorización de MPI. Se ejecuta la aplicación compilada con la biblioteca y el servidor se ejecuta con la variable DEBUG_MPI_MESSAGE puesta a 1.
Resultado esperado	Impresión del mensaje de monitorización MPI recibido.
Resultado obtenido	Satisfactorio

Tabla 30. Prueba 4

Identificador	PR-05
Objetivo	Validar la inserción de los datos del mensaje de monitorización MPI en la base de datos.
Descripción	Ejecutar la aplicación y el servidor de monitorización. Tras finalizar la ejecución de la aplicación, consultar la tabla de registros de MPI para comprobar que no está vacía.
Resultado esperado	Impresión de todos los mensajes enviados y recibidos durante la ejecución.
Resultado obtenido	Satisfactorio

Tabla 31. Prueba 5

Identificador	PR-06
Objetivo	Verificar la recepción del mensaje de monitorización del *daemon* por parte servidor.
Descripción	Se modifica el servidor para imprimir el contenido del mensaje de monitorización del *daemon*. Se ejecutan el servicio de monitorización (*daemon*) y el servidor, este último con la variable DEBUG_DAEMON_MESSAGE puesta a 1.
Resultado esperado	Impresión del mensaje de monitorización enviado por el *daemon*.
Resultado obtenido	Satisfactorio

Tabla 32. Prueba 6

Identificador	PR-07

Objetivo	Validar la inserción de los datos del mensaje de monitorización del *daemon* en la base de datos.
Descripción	Con la tabla de mensajes del *daemon* vacía o sin crear en la base de datos, ejecutar el servidor de monitorización. Ejecutar el *daemon* con 1.000 ms como frecuencia de muestreo y 1 muestra por paquete. Tras 1 segundo, terminar la ejecución del *daemon*.
Resultado esperado	Existencia de la tabla de registros del *daemon* con una fila insertada.
Resultado obtenido	Satisfactorio

Tabla 33. Prueba 7

Identificador	PR-08
Objetivo	Validar la separación en el almacenamiento de los datos inserción de los datos de monitorización del *daemon* y los datos de monitorización de MPI.
Descripción	Ejecutar todo el sistema: aplicación basada en MPI con la biblioteca desarrollada enlazada, el *daemon* y el servidor de monitorización. Terminar la ejecución del sistema al finalizar la ejecución de la aplicación “ping-pong”. Utilizando sqlite3, abrir el fichero que contiene la base de datos con el siguiente comando: “.open mpi_database.db”. A continuación, ejecutar el comando “.tables”.
Resultado esperado	Existencia de dos tablas: DAEMON_LOG y MPI_LOG.
Resultado obtenido	Satisfactorio

Tabla 34. Prueba 8

Identificador	PR-09
Objetivo	Validar la inclusión del *timestamp* junto con el resto de datos del mensaje monitorización de MPI en la base de datos.
Descripción	Tras ejecutar la aplicación “ping-pong” con la biblioteca desarrollada enlazada y el servidor de monitorización, consultar el contenido de la tabla MPI_LOG.
Resultado esperado	Existencia de la columna “TIMESTAMP” con valores de fecha y tiempo incluidos en el periodo de ejecución de la

	aplicación “ping-pong”.
Resultado obtenido	Satisfactorio

Tabla 35. Prueba 9

5.2- MATRIZ DE TRAZABILIDAD

En esta sección se muestra la matriz de trazabilidad, de manera que se pueda verificar visualmente que todos los requisitos funcionales especificados tengan al menos una prueba que valide su implementación.

	PR-01	**PR-02**	**PR-03**	**PR-04**	**PR-05**	**PR-06**	**PR-07**	**PR-08**	**PR-09**
RF-01	×								
RF-02		×							
RF-03			×						
RF-04				×					
RF-05				×					
RF-06				×					
RF-07				×					
RF-08					×				
RF-09									×
RF-10						×			
RF-11						×			
RF-12							×		
RF-13								×	

Tabla 36. Matriz de trazabilidad

5.3- EVALUACIÓN DEL SISTEMA

En esta sección se exponen y se analizan los resultados obtenidos durante la evaluación del sistema distribuido de monitorización. Como se expuso en el apartado

5.1, para las pruebas se utilizó la aplicación “ping-pong”. Dicha aplicación fue utilizada para validar el sistema desarrollado por los siguientes motivos:

- Es simple (solo dos procesos involucrados en la ejecución)
- Utiliza únicamente las funciones a monitorizar (MPI_Send y MPI_Recv)
- Es rápido (la aplicación se ejecuta completamente en menos de un minuto)

Sin embargo, para evaluar el sistema se ha optado por una aplicación más compleja. Dicha aplicación se describe en el siguiente apartado.

5.3.1.- BENCHMARKS

La aplicación que se utiliza para valorar el rendimiento del sistema está contenida en una *suite* de aplicaciones y programas diseñados y desarrollados por la NASA *“para evaluar el rendimiento de supercomputadores paralelos”* [15].

La *suite* de contiene diversos *benchmarks*, o tests para medir el rendimiento del sistema. Cada uno de estos *benchmarks* imita el procesamiento de datos y los cálculos que se dan con frecuencia en la mecánica de fluidos computacional (*Computational Fluid Dynamics* o *CFD* en inglés). En este campo en el que se utilizan los supercomputadores y la computación paralela para resolver problemas. Algunos de los *benchmarks* tratan aspectos como la ordenación de números enteros, el cálculo transformadas de Fourier tridimensionales o los *solvers* de sistemas de ecuaciones, entre otros.

Para la implementación de los *benchmarks* se utilizan diferentes modelos de computación paralela: paso de mensajes (OpenMPI), memoria compartida (OpenMP) o modelos híbridos. Cada aplicación está diseñada para medir el rendimiento en un aspecto concreto de la computación: comunicación grupal de procesos, entrada-salida paralela o acceso a memoria. Aparte de esto cada problema ofrece distintas versiones variando la cantidad de datos iniciales, y con ello la carga computacional.

5.3.2.- PREPARACIÓN DEL ENTORNO DE EVALUACIÓN

Puesto que las aplicaciones contenidas en la *suite* NPB están diseñados para ser ejecutados en supercomputadores, para la evaluación del sistema se utilizará un clúster, repartiendo la ejecución de los programas entre distintos nodos.

Las especificaciones del hardware utilizado son:

Sistema operativo	Ubuntu 16.04.2 LTS
Procesador	Intel(R) Xeon(R) CPU E5-2603 v4
Núcleos	12
Frecuencia	1.70 GHz
Memoria RAM	126 GB
Almacenamiento	1 TB

Tabla 37. Especificaciones del hardware utilizado

* Para la evaluación se utilizaron 7 equipos con las especificaciones indicadas en la tabla.

Para la evaluación del sistema se ha utilizado el *NPB* en su versión 3.2, concretamente la aplicación “Data Traffic” en dos de sus diferentes versiones: la clase C que se divide en 85 procesos y la clase D, en 171. [16]

5.3.3.- RESULTADOS OBTENIDOS

En este apartado se muestran los resultados obtenidos durante la ejecución de los *benchmarks* para la evaluación del sistema. A partir de dichos resultados se extraen conclusiones.

5.3.3.1.- Benchmark DT.C WH con 85 procesos

En esta ejecución en la que se incluye la biblioteca de monitorización de funciones MPI se han obtenido los siguientes resultados:

- Tiempo total de ejecución: 1,479 segundos
- Número total de operaciones: 87 (2 MPI_Send y 85 MPI_Recv)

```
SELECT COUNT(*) AS MPI_Recv FROM MPI_LOG WHERE strftime('%H:%M:%f', TIMESTAMP) > '12:38:00.0'
AND strftime('%H:%M:%f', TIMESTAMP) < '12:38:30.0' AND OPERATION='r';
```

MPI_Recv

85

Figura 10. Ejecución del benchmark WH con 85 procesos (MPI_Recv)

```
SELECT COUNT(*) AS MPI_Send FROM MPI_LOG WHERE strftime('%H:%M:%f', TIMESTAMP) > '12:38:00.0'
AND strftime('%H:%M:%f', TIMESTAMP) < '12:38:30.0' AND OPERATION='s';
```

MPI_Send

2

Figura 11. Ejecución del benchmark WH con 85 procesos (MPI_Send)

En la ejecución de este mismo caso sin la biblioteca de monitorización el tiempo de ejecución es de 1,447 segundos. A partir de estos resultados se puede obtener el *overhead* total introducido por la biblioteca de monitorización de funciones MPI. En este caso 32 milisegundos.

5.3.3.2.- Benchmark DT.C BH con 85 procesos

En esta ejecución en la que se incluye la biblioteca de monitorización de funciones MPI se han obtenido los siguientes resultados:

- Tiempo total de ejecución: 7,041 segundos.
- Número total de operaciones: 87 (134 MPI_Send y 131 MPI_Recv)

```
SELECT COUNT(*) AS MPI_Recv FROM MPI_LOG WHERE strftime('%H:%M:%f', TIMESTAMP) > '12:39:00.0'
AND strftime('%H:%M:%f', TIMESTAMP) < '12:40:30.0' AND OPERATION='s';
```

MPI_Recv

134

Figura 12. Ejecución del benchmark BH con 85 procesos (MPI_Send)

```
SELECT COUNT(*) AS MPI_Recv FROM MPI_LOG WHERE strftime('%H:%M:%f', TIMESTAMP) > '12:39:00.0'
AND strftime('%H:%M:%f', TIMESTAMP) < '12:40:30.0' AND OPERATION='r';
```

MPI_Recv

131

Figura 13. Ejecución del benchmark BH con 85 procesos (MPI_Recv)

Sin la biblioteca el tiempo de ejecución es de 6,795 segundos.

En esta versión de la aplicación DT, el *overhead* introducido es de 240 milisegundos.

5.3.3.3.- Benchmark DT.D BH con 171 procesos

En esta ejecución en la que se incluye la biblioteca de monitorización de funciones MPI se han obtenido los siguientes resultados:

- Tiempo total de ejecución: 3,291 segundos.
- Número total de operaciones: 87 (2 MPI_Send y 171 MPI_Recv)

```
SELECT COUNT(*) AS MPI_Recv FROM MPI_LOG WHERE strftime('%H:%M:%f', TIMESTAMP) > '12:36:00.0'
AND strftime('%H:%M:%f', TIMESTAMP) < '12:36:30.0' AND OPERATION='r';
```

MPI_Recv

46

Figura 14. Ejecución del benchmark BH con 171 procesos (MPI_Recv)

```
1 SELECT COUNT(*) AS MPI_Recv FROM MPI_LOG WHERE strftime('%H:%M:%f', TIMESTAMP) > '12:36:00.0'
2 AND strftime('%H:%M:%f', TIMESTAMP) < '12:36:30.0' AND OPERATION='s';
```

```
MPI_Recv
43
```

Figura 15. Ejecución del benchmark BH con 171 procesos (MPI_Send)

En la ejecución de este mismo caso sin la biblioteca de monitorización el tiempo de ejecución es de 3,266 segundos. En este caso el retardo incluido por la biblioteca es de 25 milisegundos.

5.3.3.4.- Benchmark DT.D WH con 171 procesos

En esta ejecución en la que se incluye la biblioteca de monitorización de funciones MPI se han obtenido los siguientes resultados:

- Tiempo total de ejecución: 6,098 s
- Número total de operaciones: 87 (2 MPI_Send y 171 MPI_Recv)

```
1 SELECT COUNT(*) AS MPI_Recv FROM MPI_LOG WHERE strftime('%H:%M:%f', TIMESTAMP) > '12:37:00.0'
2 AND strftime('%H:%M:%f', TIMESTAMP) < '12:38:00.0' AND OPERATION='s';
```

MPI_Recv
2

Figura 16. Ejecución del benchmark WH con 171 procesos (MPI_Send)

```
1 SELECT COUNT(*) AS MPI_Recv FROM MPI_LOG WHERE strftime('%H:%M:%f', TIMESTAMP) > '12:37:00.0'
2 AND strftime('%H:%M:%f', TIMESTAMP) < '12:38:00.0' AND OPERATION='r';
```

MPI_Recv
171

Figura 17. Ejecución del benchmark WH con 171 procesos (MPI_Recv)

En la ejecución de este mismo caso sin la biblioteca de monitorización el tiempo de ejecución es de 6,101 s. En este caso el retardo incluido por la biblioteca es de 25 milisegundos.

5.3.4.- CONCLUSIONES

Tras ejecutar los distintos *benchmarks*, se ha podido comprobar que la biblioteca desarrolla para capturar las llamadas a MPI_Send y a MPI_Recv cumple su cometido en cuanto a que no introduce retardos considerables si se tiene en cuenta la cantidad de operaciones que se realizan en tan corto espacio de tiempo.

6.- PLANIFICACIÓN Y PRESUPUESTO

En este capítulo se especifican las distintas etapas e hitos en los que se ha dividido la realización de este Trabajo de Fin de Grado. Se desglosan además los costes derivados de la realización de este proyecto.

6.1.- PLANIFICACIÓN

En esta sección se detallan cada una de las actividades en las que se ha distribuido la realización del proyecto. Se indican, aproximadamente, las fechas de inicio y fin, y las horas dedicadas a cada una las tareas.

Tarea	Inicio	Fin	Duración (horas)
Planificación	18/10/17	23/10/17	10
Análisis	6/11/17	15/11/17	15
Diseño	16/11/17	19/12/17	20
Implementación	3/1/18	10/4/18	70
Pruebas	12/4/18	24/5/18	40
Evaluación	24/5/18	18/6/18	15
Revisión	10/6/18	18/6/18	10
Documentación	17/11/18	18/6/18	80

Tabla 38. Seguimiento de tareas

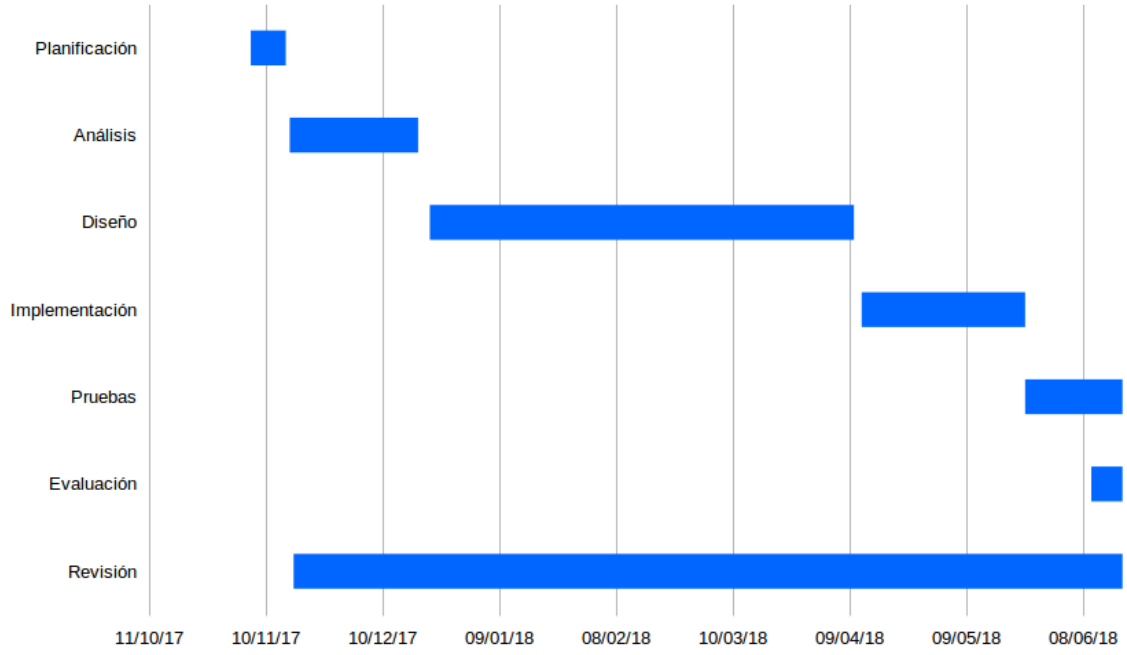

Figura 18. Diagrama de Gantt (planificación)

6.2.- PRESUPUESTO

En los apartados sucesivos se detallarán los costes estimados, clasificándolos en tres grandes bloques: costes derivados del hardware, software y recursos humanos.

6.2.1.- HARDWARE

En cuanto al hardware se calcula el coste imputable teniendo en cuenta el coste de la maquinaria, el tiempo estimado de amortización y el periodo de utilización durante el desarrollo del Trabajo de Fin de Grado.

Material	**Coste unitario**	**Periodo de amortización**	**Periodo de utilización**	**Coste amortizado**
Portátil HP Pavilion A10 5745M	690,00 €	60 meses	7	80,50 €
Total	-	-	-	80,50 €

Tabla 39. Coste del hardware

Para llevar a cabo la evaluación del sistema distribuido se utilizó un clúster prestado por el grupo de investigación ARCOS, perteneciente a la Universidad Carlos III.

6.2.2.- SOFTWARE

Software	Coste unitario	Coste amortizado
Google Docs, Google Drive	0,00 €	0,00 €
SQLite	0,00 €	0,00 €
MPICH	0,00 €	0,00 €
Ubuntu 16.04 LTS	0,00 €	0,00 €
Total	0,00 €	0,00 €

Tabla 40. Coste del software

6.2.3.- RECURSOS HUMANOS

A continuación se especifican los participantes y la labor desempeñada durante la realización de este Trabajo de Fin de Grado.

Puesto	Participante
Analista programador	Edgar Mijero Bonde

Tabla 41. Participantes en el proyecto

Una vez especificados los participantes en el desarrollo del sistema, se calculan los costes en función del número de horas dedicadas por cada uno ellos.

Participante	Horas	Coste (€/h)
Analista programador	260	22,75 €
Total	260	5.915,00 €

Tabla 42. Coste asociado a recursos humanos

6.2.4.- TOTAL

Tras haber calculado el coste por apartados en la siguiente tabla se muestra el coste total derivado de la realización de este proyecto:

Concepto	Coste
Hardware	80,50
Software	0,00 €
Recursos humanos	5.915,00 €
Total sin IVA	5.995,50 €
Riesgo estimado (20%)	1199,10 €
Beneficio estimado (30%)	1798,65 €
IVA (21%)	1.259,06 €
Total con IVA	**7.254,56 €**

Tabla 43. Coste total

7.- CONCLUSIONES Y TRABAJOS FUTUROS

En este capítulo se valora la medida en la que los objetivos definidos al principio de este documento han sido alcanzados, se presentan las dificultades que surgieron durante la realización de este trabajo y se proponen mejoras del sistema desarrollado, así como posibles líneas de investigación futuras.

7.1.- CONCLUSIONES

El objetivo final de la realización de este proyecto era la implementación de un sistema distribuido capaz de monitorizar la ejecución aplicaciones paralelas basadas en MPI, de modo que posteriormente se pueda analizar el rendimiento de dichas aplicaciones. Se considera que dicho objetivo ha sido alcanzado satisfactoriamente.

Profundizando en las tareas en las que se dividió la consecución del objetivo principal:

- Se implementado un sistema de monitorización que apenas interfiere con la ejecución de la aplicación a analizar, puesto que únicamente hay que compilarla con la biblioteca desarrollada para la captura de métricas relativas a MPI.
- Relativo al diseño, se considera que para la biblioteca desarrollada se ha obtenido una implementación fácilmente extensible en caso de requerir otro tipo de datos relativos a MPI para el análisis.
- Se han conocido detalles de la implementación de la especificación MPI, concretamente la implementación de MPICH, llegando a extender dicha implementación.
- Se ha extendido la funcionalidad de una aplicación existente, integrando dicha aplicación en el sistema distribuido desarrollado.

7.2.- IMPACTO SOCIOECONÓMICO

Como parte de este Trabajo de Fin de Grado se ha implementado un sistema distribuido que permite analizar el rendimiento en aplicaciones basadas en MPI.

Dado que el sistema desarrollado tiene un *scope* muy limitado y específico (lista reducida de funciones MPI analizadas, métricas a obtener no configurables), no se

espera obtener ningún beneficio económico derivado de su desarrollo, por lo que dicho sistema no será comercializado.

En cuanto a la distribución de la herramienta desarrollada, estará disponible como software libre. De este modo se impulsa la mejora y la extensión de las funcionalidades del sistema desarrollado.

Otro de los motivos por los que distribuirá el sistema como software libre es que todas las herramientas software utilizadas durante su implementación y validación están distribuidas como software libre. Concretamente:

- MPICH: distribuida bajo una licencia basada en licencias BSD (*Berkeley Software Distribution* [17]).
- SQLite: esta librería pertenece al dominio público por lo que no requiere licencia alguna para ser utilizada o modificada de cualquier manera [18].
- NASA Parallel Benchmark: (*NPB*) está distribuida bajo una licencia de software libre emitida por la NASA (*NASA Open Source Agreement v1.3* [19]) ().

Dado el creciente interés y el aumento de la investigación en computación paralela, se considera que, pese al limitado alcance del sistema implementado, éste puede servir de base de cara a futuras herramientas.

7.3.- TRABAJOS FUTUROS

En esta sección se presentarán posibles mejoras a realizar en el sistema desarrollado y nuevas funcionalidades a implementar.

Como se expuso previamente, el sistema desarrollado tiene un *scope* bien definido por lo que una de las mejoras inmediatas que se podrían realizar sobre el sistema es, por ejemplo, aumentar el número de funciones de MPI cuya información se quiere obtener. Esto permitiría analizar aplicaciones basadas en MPI más complejas: aplicaciones con comunicación grupal (*reduce*, *broadcast*, etcétera) o aplicaciones que trabajasen con ficheros a través de MPI, entre otras.

Otra funcionalidad que se podría añadir al sistema sería dotarlo de interfaz gráfica de modo que los datos obtenidos gracias a la monitorización se mostrasen en la interfaz. De esta manera se podrían detectar visualmente puntos en los que aumenta el tráfico de mensajes, cuellos de botella u otras situaciones susceptibles de análisis u optimización.

Finalmente, la última línea de desarrollo propuesta es la inclusión de algunas de las funcionalidades del *daemon* en la biblioteca MPI. Con esta combinación de funcionalidades las métricas relativas al estado de la máquina durante la ejecución de las aplicaciones basadas en MPI sería más precisas, ya que las métricas se generarían basándose en eventos (invocaciones de funciones de MPI) en lugar de ser generadas de

manera síncrona (intervalo de muestreo). Un posible inconveniente que tiene esta última propuesta es que añadir la obtención de métricas referentes al estado del sistema introduciría *overhead*, lo que influiría en la ejecución normal de la aplicación a analizar.

Como última propuesta a largo plazo, y más en el ámbito de la investigación, sería útil poder extraer patrones a partir de las operaciones realizadas y los datos obtenidos. Esta categorización de las ejecuciones potenciaría el análisis, permitiendo ofrecer informes con los problemas detectados o incluso posibles optimizaciones.

8.- PARALLEL PROGRAMMING AND ITS USES

In the last couple of years we have experienced an exponential growth in the number and variety of electronic devices. The number of smartphone users keep rising. The development of smarthouses and the automation of buildings and cities have supposed an increase in the number of IoT (*Internet of Things*) devices.

This increase in the quantity of devices has led to a growth in the quantity of data generated by those devices. In addition to that, the proliferation of social networks such as Twitter, Facebook, Instagram or Snapchat, has contributed to a need of more processing power needed to work with all the information and data generated (images, video, audio, text messages, etc).

This rise of data has made the requirements regarding processing power bigger. Moore's Law roughly states that computer processor speed, and therefore processing power, doubles every two years. This prediction or trend has proven to be accurate since the law was formulated in 1965, but in the last years, around 2015 its has reached a point of stagnation. This stagnation is due to the fact that we are reaching the maximum transistor density per processor.

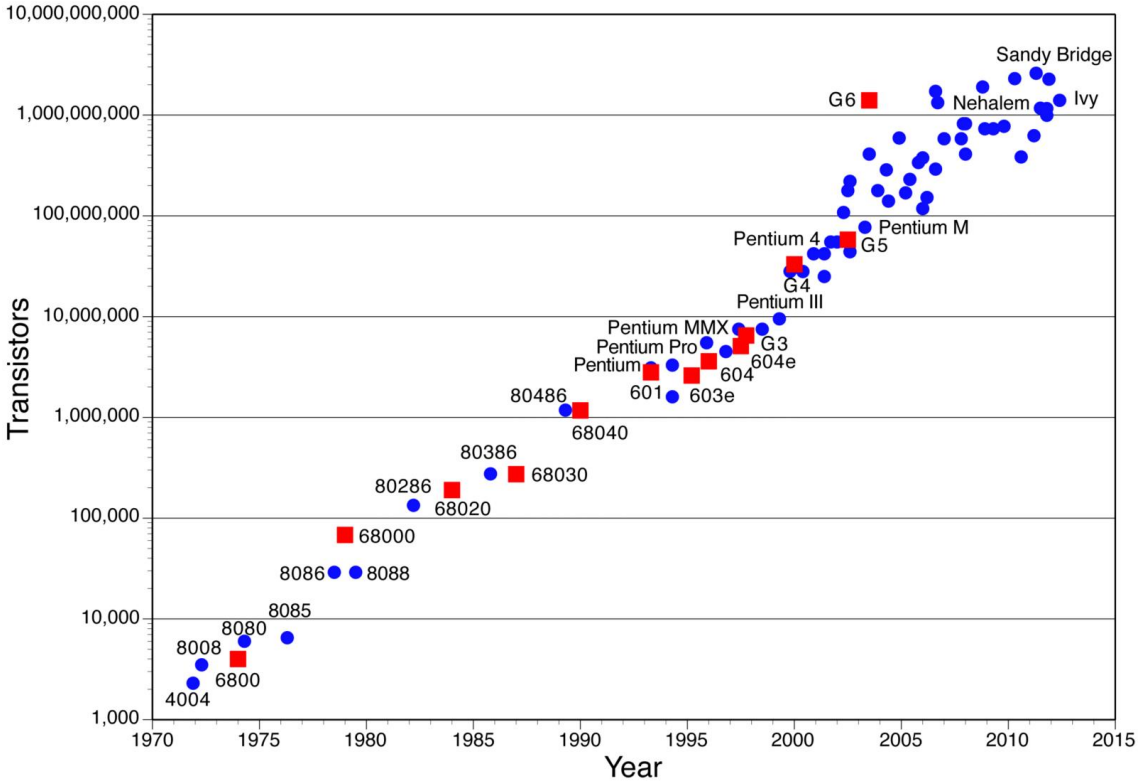

Figura 19. Ley de Moore [20]

This cap in the growth of processing power by the addition of transistors to processors, has led to the investigation and promotion of alternative improvements to keep gaining processing power. One of those alternatives is parallel computing.

Since Moore's Law no longer predicts nor represent accurately the growth of processing power, the sequential approach that had been (and is still being) used, has to gradually open up space for the parallel approach, because is the one that offers more processing power growth rate.

8.1.- PARALLEL COMPUTING

The parallel computing approach consists on dividing the execution of an application or any big computation task in smaller tasks that run simultaneously. Ideally the "subtasks" are independent so the absence of race conditions is guaranteed.

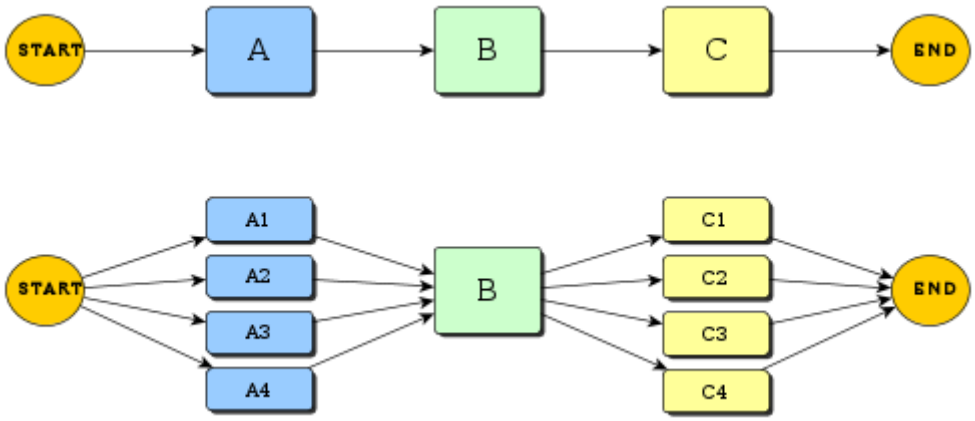

Figura 20. Ejecución secuencial vs ejecución paralela [21]

In the previous figure we can observe the representation of both approaches: the sequential and the parallel one.

In the sequential approach, the application is executed by a single process from start to end with the occasional *spawn* of threads for specific tasks. In contrast, in parallel programs the sequential parts, since everything can't be parallelized, are the parts where task assignment, computation results gathering, aggregation and control takes place.

Parts A and C on the figure might represent a computation intense section on the program. Those parts are susceptible of optimization by partitioning them into execution chunks that run at the same time. The time spent processing the □□ tasks generally is lesser than the time spent on task A on the sequential approach, ideally $\frac{□}{□}$.

8.2.- PARALLEL COMPUTING MODELS

One of the main concerns of parallel computing programmers is the way in which tasks are divided and assigned to cores. Since not all applications, and surely not all the parts that compose a program, are parallelizable, programmers use heuristics to determine how a program can be implemented to take advantage of parallelism.

The use of these heuristics has led to classifying parallel programs by the techniques used to parallelize them. These techniques have been named as parallel programming models.

Those models are explained in the following sections.

8.2.1- SIMD (Single Instruction Multiple Data)

Often called "vectorization" is is the application of a single instruction to multiple sets of data at the same time. This technique is widely used in fields such as multimedia, signal processing and matrix operations.

The basic premises for using this technique is the independence of data, that is, applying the single instruction to disjunct sets of data. In this model, the main concepts are the data assignment and the "loop unrolling".

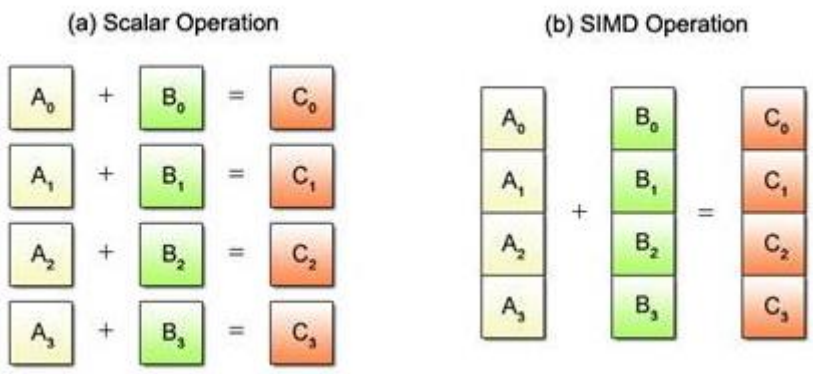

Figura 21. Operación escalar vs SIMD [22]

In the figure above we have a comparison of two approaches to solve the addition of two matrices.

In the sequential approach we would probably apply a single operation to a single set of data, iterating to complete the addition of all the components of the matrices. Something in the lines along:

```
for(i = 0; i < array_size; i++){
    c[i] = a[i] + b[i];
}
```

Instead in SIMD approach we make use the two concepts presented:

- Data independence: the data to which the single instruction is applied its independent for each processing unit. In matrix addition we add the values of the cell based on their indexes so there's no possibility for applying the same instruction twice to the same set of data on an iteration.
- Loop unrolling: this is the step where we switch from a sequential approach to a SIMD one. We take an instruction or operation that was being applied to a unique set of data (one matrix position per loop iteration) and transform apply it to many (add all the position on a single "iteration").

With that said, the resulting for the "unrolled loop" would be:

```
c[0] = a[0] + b[0];
c[1] = a[1] + b[1];
c[2] = a[2] + b[2];
c[3] = a[3] + b[3];
```

8.2.2.- Shared memory

In this model processors may or not apply the same operations to different sets of data. This approach it's one of the most common parallel computing paradigms.

The way in which we proceed is by declaring sections of the program as parallel, where we assign tasks to processors and establish memory scopes for the processes. Inside parallel sections, we can also declare critical sections where an operations presumably modifies a shared variables by all the processes in which a task was divided.

Even though it's possible to declare variables as private or shared by processes, the memory where those variables reside is shared by all processes. Therefore there has to be a mechanism to synchronize processes and control the access to certain regions of memory. For that we use barriers and locks.

The main challenge with this approach is “data scoping” or how to distribute the data between processes in a way that racing conditions and overhead are diminished while increasing the speedup.

The most used API for this approach is OpenMP [23]. It bases its execution on the “fork-join” method [24], in which a tasks is divided in n subtasks and the subtasks are executed on n threads spawned by the main thread. Once the computation of subtasks is done by all the threads, their results are “joined” or aggregated on the main thread.

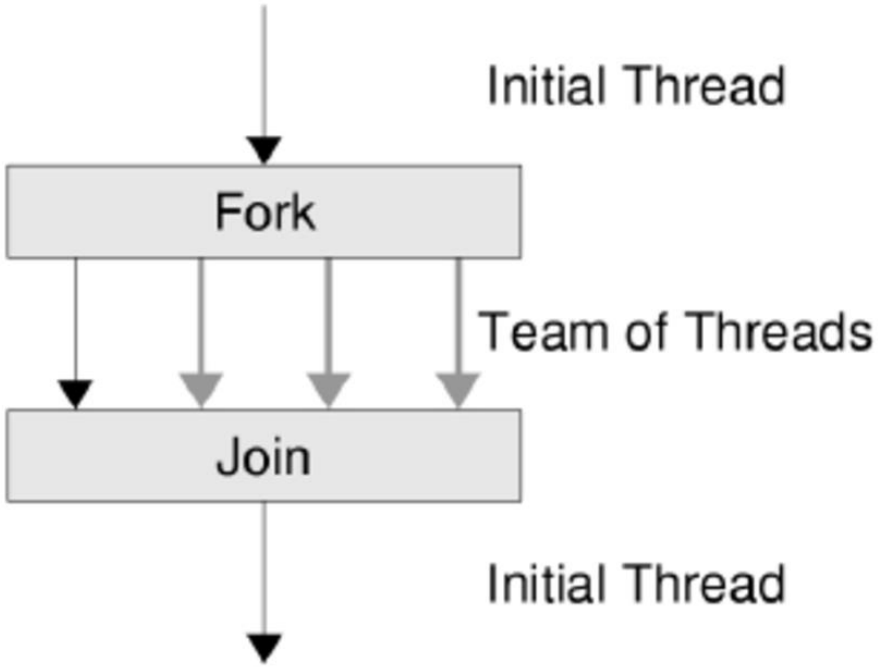

Figura 22. Fork-join

8.2.3.- MIMD (Multiple Instruction Multiple Data)

This paradigm is specially suitable to an architectura where each processor has its own memory. The main challenge this approach faces arises when those processors have to share or exchange information that they have locally.

This approach doesn’t share the main concern in shared memory parallel programs, “data scoping”, since the memory is distributed and each processor has its own memory. Conversely, there is no way of declaring shared variables, and par extension critical zones to hold and aggregate the results on the computations performed by each processor.

All the data gathering and sharing between processes is achieved by message passing. Also not all processors execute the same instructions to the data, processors may apply certain operations to the data based on some conditions or constraints.

The main standard for this parallel computing model is the MPI standard which is explained and mentioned throughout the whole project.

8.2.4.- GPGPU (General Purpose Graphics Processing Units)

The reasoning behind this approach is the use of GPUs (graphics processing units) to process information as if it was images or graphics data. Since GPUs possede a large number of ALUs (arithmetic logic units), this approach is specially suitable for applications can easily be partitioned, ideally in form of vectors, this way data processing can happen simultaneously.

The main concern programmers have when working with GPGPU is preparing the data so it can be treated as images or graphics data [25]. Once that hurdle is overcomed, programmers are able to exploit all the benefits GPUs offer.

The main tool used by developers to program GPUs is CUDA, a programming model developed by the technology company NVIDIA.

8.3.- FIELDS WHERE PARALLEL COMPUTING IS USED

In the following paragraphs, different examples of parallel computing applications are mentioned. The problems that are being solved using parallelism are explained in a succinct way.

Astrophysics

N-Body problem

Parallel computing is used in astrophysics to model an important problem. This problem is the "N-Body problem" and it represents the evolution of a system in which its participants survive or die based on their interactions with the rest of components of the system. For each component of the system the sum of forces that the rest of elements exert is computed. With those results the change in movement (speed and direction) of each component is applied for the next iteration. The parallel part in this problem is the computation of the sum that each component exerts to another. [26]

Finite Element Analysis

This computing paradigm is also used in Finite Elements Analysis (FEA) or Finite Element Method (FEM) [27]. The FEA technique consists on breaking down a complex system or structure for which we cannot easily compute the change in value of a certain attribute in presence of stimuli. The initial system gets broken into smaller parts that we are capable of model with more accurately. Then, for each "node", a set of equations is defined to calculate the value of property we wanted to compute for the initial system. Nodes and their equations (generally differential) get combined into "mesh equations", combinations of nodes that offer an accurate representation of a part of the system. Mesh equations are solved simultaneously taking advantage of parallelism and the results are combined to gain a more accurate representation of the physical properties of the system.

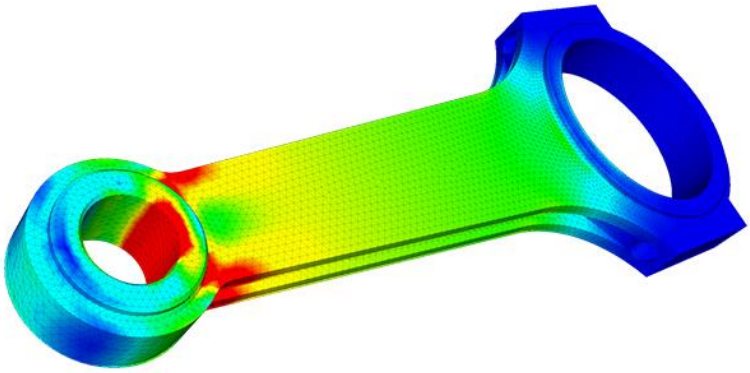

Figura 23. F.E.A [28]

Medicine

Parallel computing is also used in medicine in areas like DNA and molecules analysis and classification.

One of the frequent task in which parallel computing is involved is in cell classification aided by machine learning models or algorithms. Parallel programs are fed with large amounts of images to process them and extract their main properties used to group them based on those properties.

Another use is on DNA sequence analysis. Due to the large size of single person genetic information and the variety of patterns, analyzing that info requires of tremendous processing power. The current approach followed in bioinformatics currently is the use of big data methods, such as MapReduce.

MapReduce is a programming model that takes a large set of data, divide it into smaller pieces and assign it to tasks. This process is called *mapping* data to taks. After having performed the set of operations specified to their assigned data, the results returned by the tasks are sorted or shuffled in some way, and then passed to a reduce operation which can consist of filter, sum or any other type of aggregation.

Mathematics

Mathematics is another field in which parallel computing can be helpful, specially in most types of equation solvers and most kinds of matrix operations. This is because matrices are suitable candidates to be processed in parallel since generally they can be divided in vectors which are a great data structured to work with in parallel computing.

Image processing

Images processing is another field where parallel computing results in notable gains in performance. Images are composed of pixels, so for each image we have a matrix, which can also be divided into vectors. With this setting, it is possible to apply any of the models mentioned before.

Even if any of the methods can be used in this case, the SIMD and the GPGPU are the ones that fit better with the data available.

GPGPU might be the most suitable since GPU were designed originally to speed image or graphics data processing. But SIMD is also an interesting approach since usually image operations are evenly and uniformly applied. This means that we can execute the same operations to to every row or column of the pixel matrix.

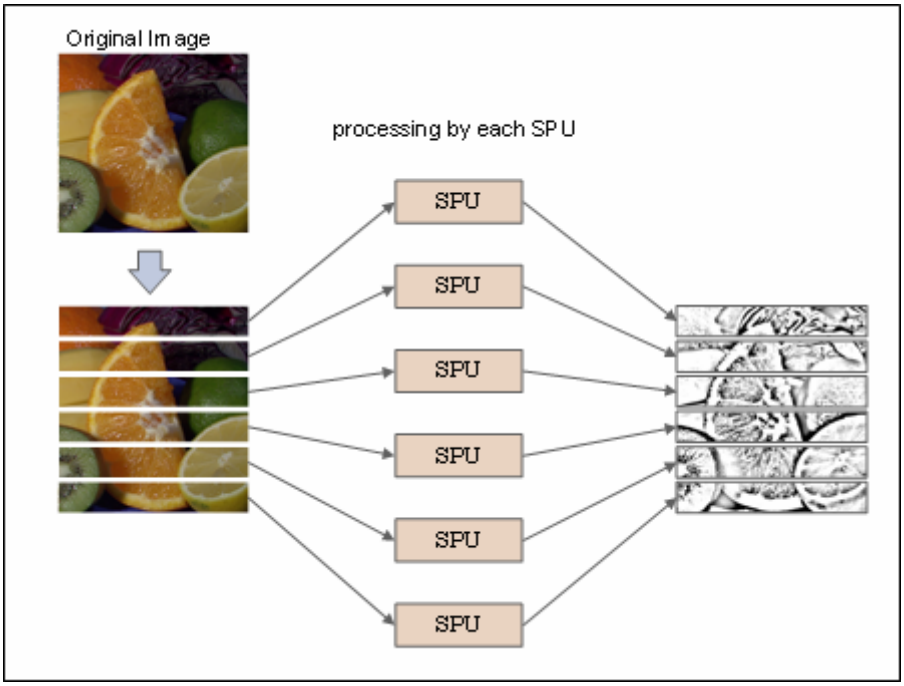

Figura 24. Procesamiento de imágenes paralelo

In the image above the way the data is partitioned and assigned to each processing unit, which perform the same filtering operation simultaneously to different sets of data, in this case rows of the original image (matrix of pixels).

8.4.- CONCLUSION

In conclusion, an effort has to be made to allow programmers to seamlessly switch from the sequential approach that has been predominantly until now to a more parallel and optimized approach.

It is not going to be easy since not all the applications, programs nor functionalities can be parallelized. But there has to be a shift towards optimization taking into account that the quantity of data keeps increasing at astonishing levels. So, to catch up, or to make that transition as smooth as possible, new tools to facilitate parallel programming have to be build.

In fact that is one of the goals of this project, to contribute to space of tools for programmers to help them not only to write parallel applications, but also to measure and assess the performance of the programs they develop.

To sum everything up, programmers and computer scientists have to start embracing the advantages that parallel programming offers even if its techniques or approaches are not widely applicable. Most of the times the speedups and gains obtained when using a parallel approach offsets the hardships that oftentimes supposes programming with the parallel paradigm in mind.

REFERENCIAS Y BIBLIOGRAFÍA

Barney, Blaise, 2017. Introduction to parallel computing. En: *Lawrence Livermore National Laboratory* [en línea]. Disponible en: https://computing.llnl.gov/tutorials/parallel_comp/ [consulta: 18 junio 2018]

Gropp, William D., 1998. An introduction to MPI: parallel programming with the Message Passing Interface. En: *Argonne National Laboratory* [en línea]. Disponible en: http://www.mcs.anl.gov/research/projects/mpi/tutorial/mpiintro/ppframe.htm [consulta: 18 junio 2018]

Foster, Ian, 1995. Data parallelism. En: *Mathematics and science | Argonne National Laboratory* [en línea]. Disponible en: http://www.mcs.anl.gov/~itf/dbpp/text/node83.html [consulta: 18 junio 2018]

Eijkhout, Victor, 2016. MPI topic: Communicators. En: *Texas Advanced Computing Center* [en línea]. Disponible en: http://pages.tacc.utexas.edu/~eijkhout/pcse/html/mpi-comm.html [consulta: 18 junio 2018]

Barney, Blaise, 2017. Message Passing Interface (MPI). En: *Lawrence Livermore National Laboratory* [en línea]. Disponible en: https://computing.llnl.gov/tutorials/mpi/ [consulta: 18 junio 2018]

Ritchie, Dennis. Christory. En: *Bell Labs* [en línea]. Disponible en: https://www.bell-labs.com/usr/dmr/www/chist.html [consulta: 18 junio 2018]

[1] https://bioinfomagician.wordpress.com/2013/11/11/parallel-computing-introduction-to-mpi/ [consulta: 18 junio 2018]

[2] http://www.training.prace-ri.eu/uploads/tx_pracetmo/bekas_PRACE_hybrid.pdf [consulta: 18 junio 2018]

[3] https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf [consulta: 18 junio 2018]

[4] http://wgropp.cs.illinois.edu/courses/cs598-s16/lectures/lecture34.pdf [consulta: 18 junio 2018]

[5] https://www.open-mpi.org [consulta: 18 junio 2018]

[6] https://www.mpich.org/ [consulta: 18 junio 2018]

[7] https://software.intel.com/en-us/intel-mpi-library [consulta: 18 junio 2018]

[8] https://www.intel.es/content/www/es/es/high-performance-computing-fabrics/omni-path-driving-exascale-computing.html [consulta: 18 junio 2018]

[9] https://tools.bsc.es/paraver/ [consulta: 18 junio 2018]

[10] https://www.cs.uoregon.edu/research/tau/home.php [consulta: 18 junio 2018]

[11] https://hpc.llnl.gov/software/development-environment-software/mpip [consulta: 18 junio 2018]

[12] http://www.mcs.anl.gov/research/projects/perfvis/software/MPE/ [consulta: 18 junio 2018]

[13] https://software.intel.com/en-us/parallel-studio-xe https://software.intel.com/en-us/intel-vtune-amplifier-xe/details [consulta: 18 junio 2018]

[14] https://www.sqlite.org/mostdeployed.html [consulta: 18 junio 2018]

[15] https://www.nas.nasa.gov/publications/npb.html [consulta: 18 junio 2018]

[16] https://www.nas.nasa.gov/publications/npb_problem_sizes.html [consulta: 18 junio 2018]

[17] http://git.mpich.org/mpich.git/blob/HEAD:/COPYRIGHT [consulta: 18 junio 2018]

[18] https://www.sqlite.org/copyright.html [consulta: 18 junio 2018]

[19] https://opensource.org/licenses/nasa1.3.php [consulta: 18 junio 2018]

[20] https://medium.com/@DJohnstonEC/johnston-s-law-quantified-f1a4d93bbc19 [consulta: 18 junio 2018]

[21] http://xmipp.cnb.csic.es/twiki/bin/view/Xmipp/ParallelProgramming [consulta: 18 junio 2018]

[22] https://johanmabille.github.io/blog/2014/10/09/writing-c-plus-plus-wrappers-for-simd-intrinsics-1/ [consulta: 18 junio 2018]

[23] https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf [consulta: 18 junio 2018]

[24] https://www.researchgate.net/figure/The-OpenMP-fork-join-shared-memory-programming-model-An-initial-master-thread-replicates_fig1_220307944 [consulta: 18 junio 2018]

[25] https://www.tacc.utexas.edu/documents/13601/88790/8Things.pdf [consulta: 18 junio 2018]https://developer.nvidia.com/cuda-zone

[26] http://worldcomp-proceedings.com/proc/p2012/FEC8026.pdf [consulta: 18 junio 2018]

[27] https://www.manortool.com/finite-element-analysis [consulta: 18 junio 2018]

[28] https://www.simscale.com/blog/2016/10/what-is-finite-element-method/ [consulta: 18 junio 2018]

http://www.cs.nthu.edu.tw/~ychung/slides/para_programming/slides2.pdf [consulta: 18 junio 2018]

ANEXO I

LEGISLACIÓN

Ley Orgánica 15/1999, de 13 de diciembre, de Protección de Datos de Carácter Personal (BOE núm.298, de 14 de diciembre de 1999).

REQUISITOS DE SISTEMA

Para poder ejecutar el sistema distribuido de monitorización es necesario:

- Instalar MPICH en su versión 3.2.1
- Para el *daemon*, es necesario instalar CUDA, preferentemente en su versión 9.0

EJECUCIÓN DE LOS BENCHMARKS

Para ejecutar los *becnhmarks* se siguieron los siguientes pasos:

- Crear un fichero llamado *machines* con las direcciones IP o los *hostnames* de los nodos entre los que se ejecutará la aplicación.

```
compute-11-1
compute-11-2
compute-11-4
compute-11-5
compute-11-6
compute-11-7
compute-11-8
```

Tabla 44. Fichero de configuración de los nodos

- Compilar y ejecutar el *benchmark* a ejecutar de la siguiente manera:

Compilación	`make <benchmark> NPROCS=<núm. procesos> CLASS=<versión>`
Ejecución	`mpiexec -n <núm. procesos> -env SERVER_IP <ip> -env SERVER_PORT <puerto> -f machines ./<benchmark>`

Tabla 45. Instrucciones para la compilación y ejecución de benchmarks